Programmation de Processeurs Massivement Parallèles

Hubert Thibault
DS Improve SCRL

Table des Matières

Orateur

- Hubert Thibault
- Diplômé de l'ULB en 2007 – Licence en Informatique
- 2008 - 2011: Programmeur R&D chez DS Improve
  - Projets Principaux:
    - CLEVER : Moteur 3D pour le streaming de contenu en temps-réel
    - HAEVA : Moteur 3D pour le rendu architectural temps-réel

Introduction

- La Loi De Moore
- Augmentation Des Performances Séquentielles Finies
- Nouvelles Utilisations Des Transistors
- Processeurs Massivement Parallèles
- La « Nouvelle » Loi De Moore
- Modèle De Calcul Hétérogène
- Generic Multicore Chip
- Generic Manycore Chip
- Motivation
- Disponibilité

La Loi de Moore (Paraphrasée)

- *« Le nombre de transistors sur un circuit intégré double tout les deux ans. »*

  - Gordon E. Moore

Augmentation Des Performances Séquentielles Finie

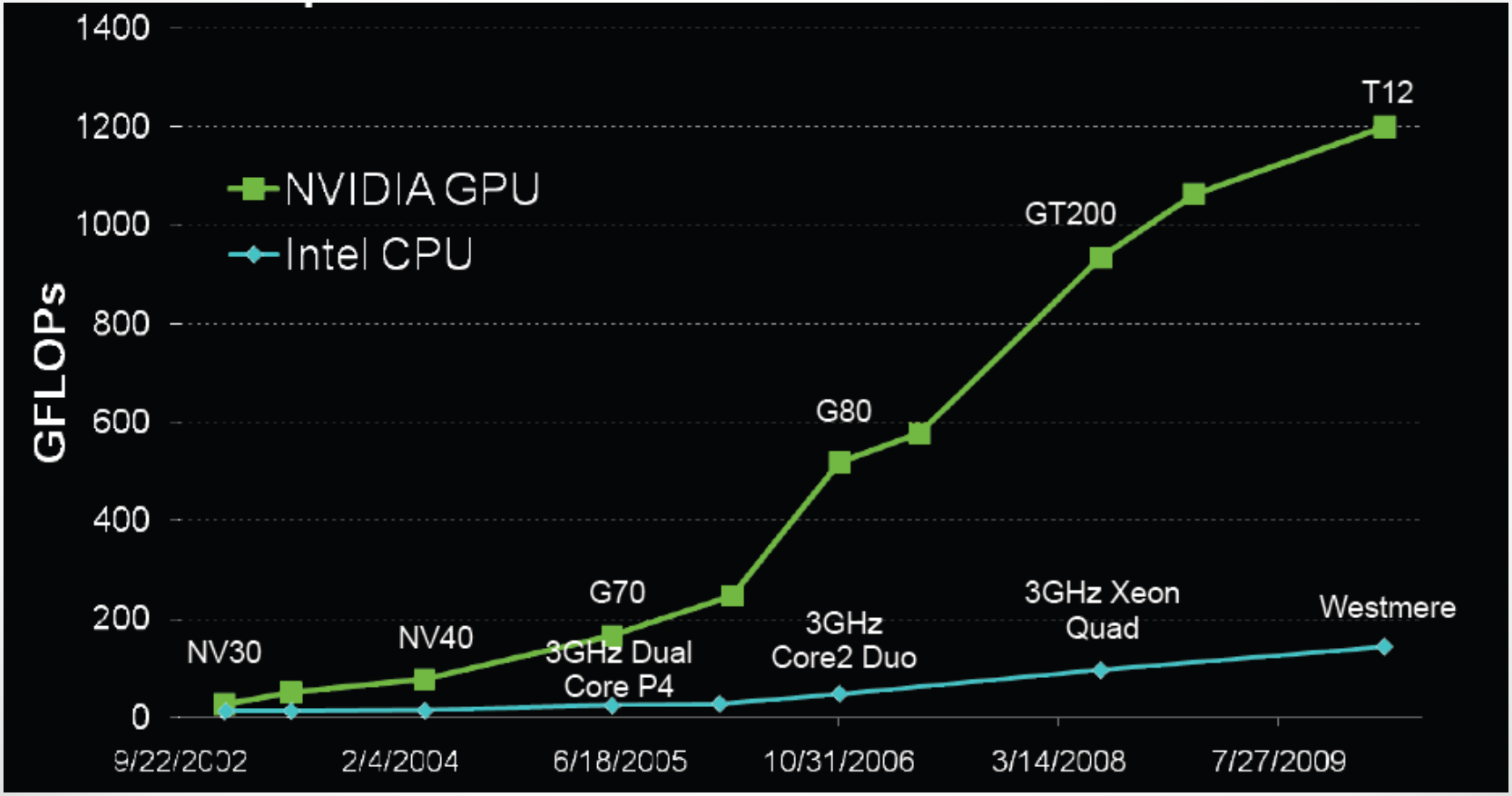
- On ne peut pas continuer à augmenter les fréquences des processeurs
  - Pas de puces à 10 GHz
- On ne peut pas continuer à augmenter la consommation de nos machines
  - Dissipation de chaleur !
- On peut continuer à augmenter la densité des transistors
  - ~ Loi de Moore

Nouvelles Utilisations des Transistors

- Parallélisme au niveau des **instructions**
  - Out-of-order execution, speculation, …
- Parallélisme au niveau des **données**
  - Vector units, SIMD execution, …
  - Exemples : SSE, Cell SPE, GPU
- Parallélisme au niveau des **threads**
  - Multithreading, multicore, manycore
  - Exemples : Intel CoreI7, AMD Phenom, Sun Niagara, STI Cell, NVIDIA Fermi, …

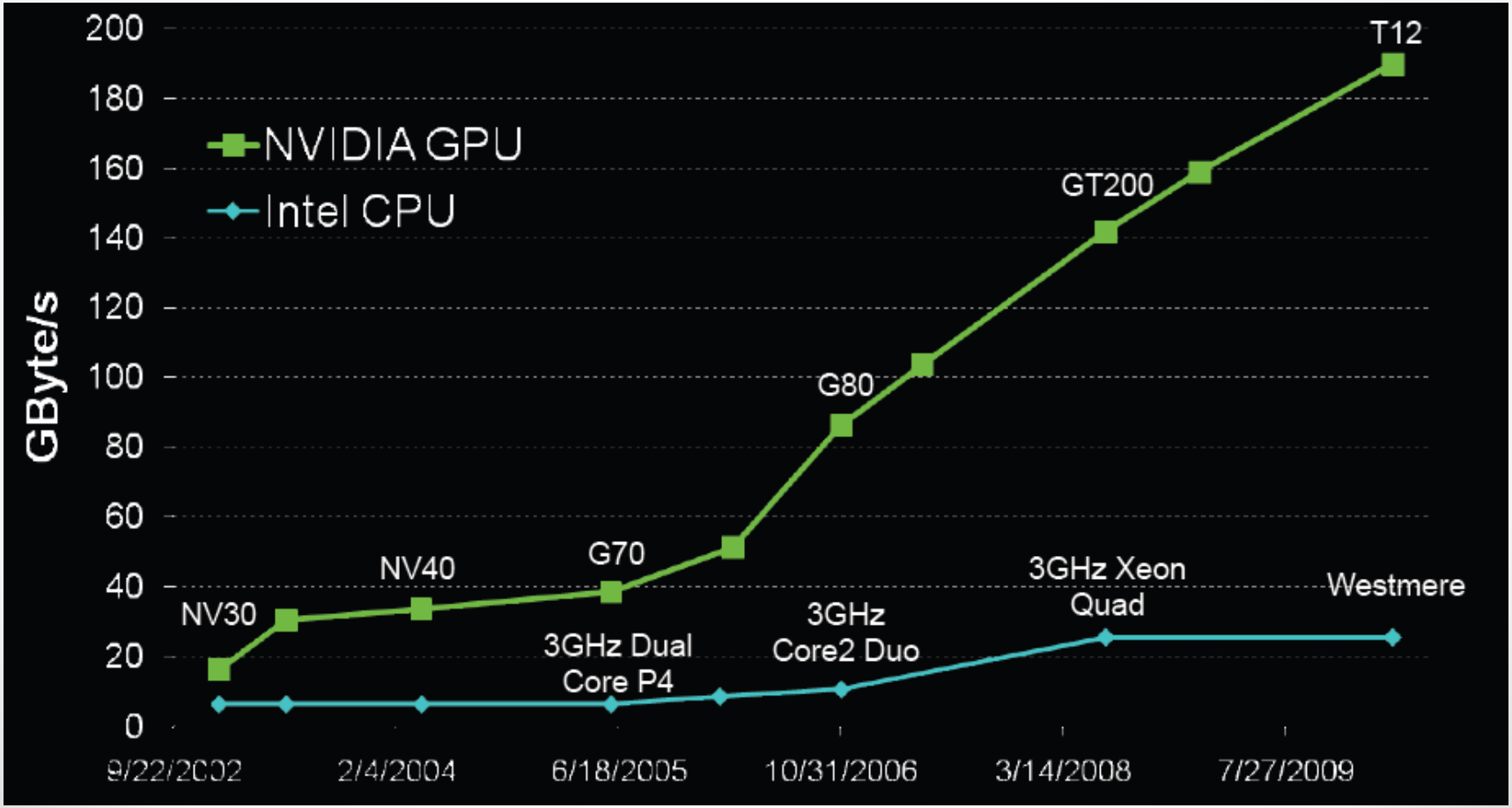
Processeurs Massivement Parallèles

- TFLOPs vs. 100 GFLOPs



Processeurs Massivement Parallèles

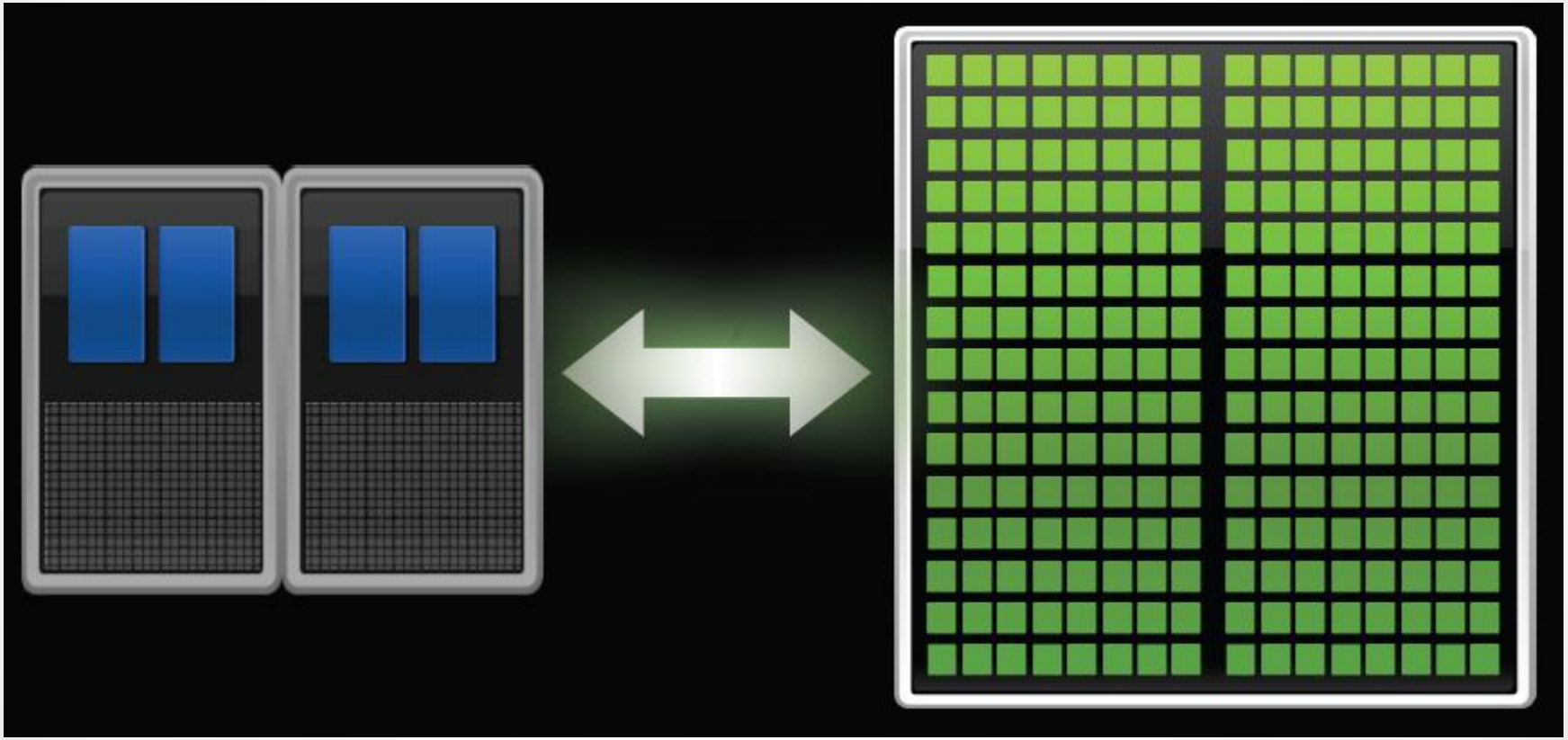
- Bande passante : ~ 10x



La « Nouvelle » loi de Moore

- Les ordinateurs ne vont pas de plus en plus vite, ils deviennent plus *larges*
- Il faut repenser nos algorithmes pour qu'ils soient parallèles
- Le parallélisme sur les données est la solution la plus extensible :
  - Sinon : Refactorisation du code pour 2 cœurs, 4 cœurs, 8 cœurs, 16 cœurs,…
  - Toujours plus de données que de cœurs → Construire les calculs autours des données

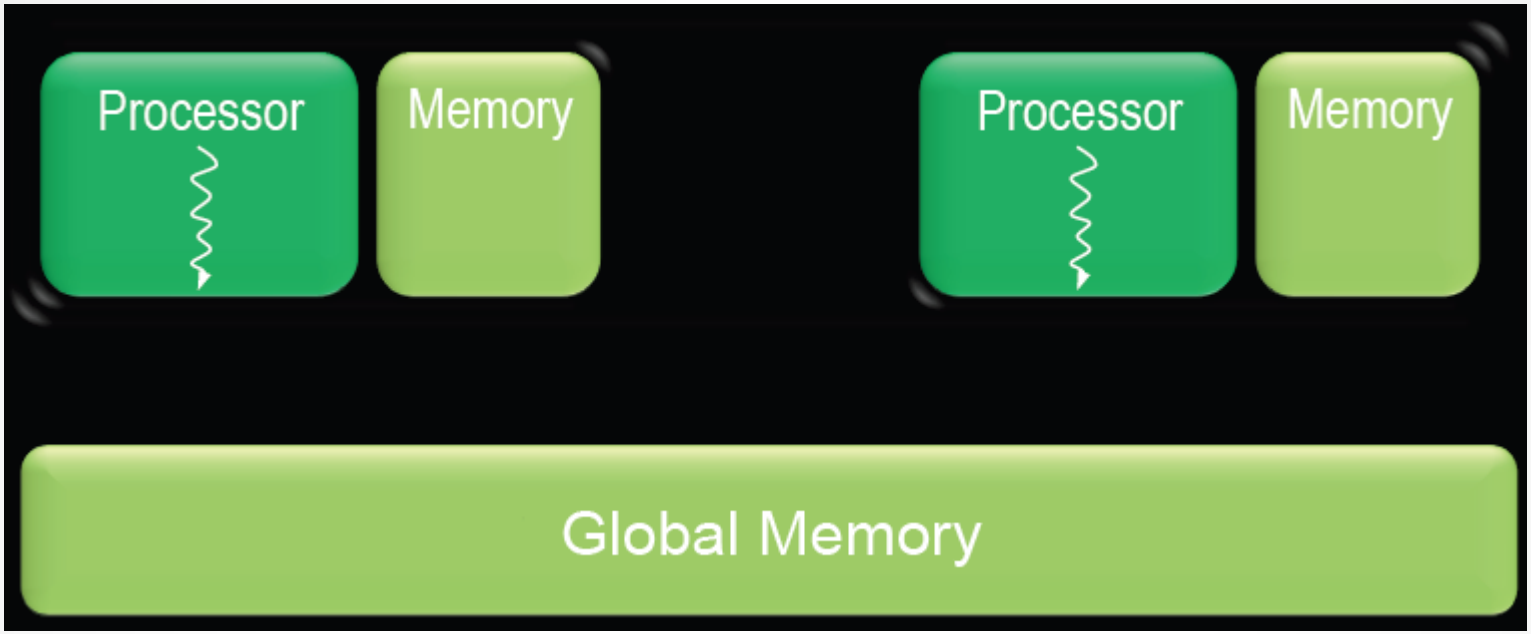
Modèle De Calcul Hétérogène



Multicore CPU

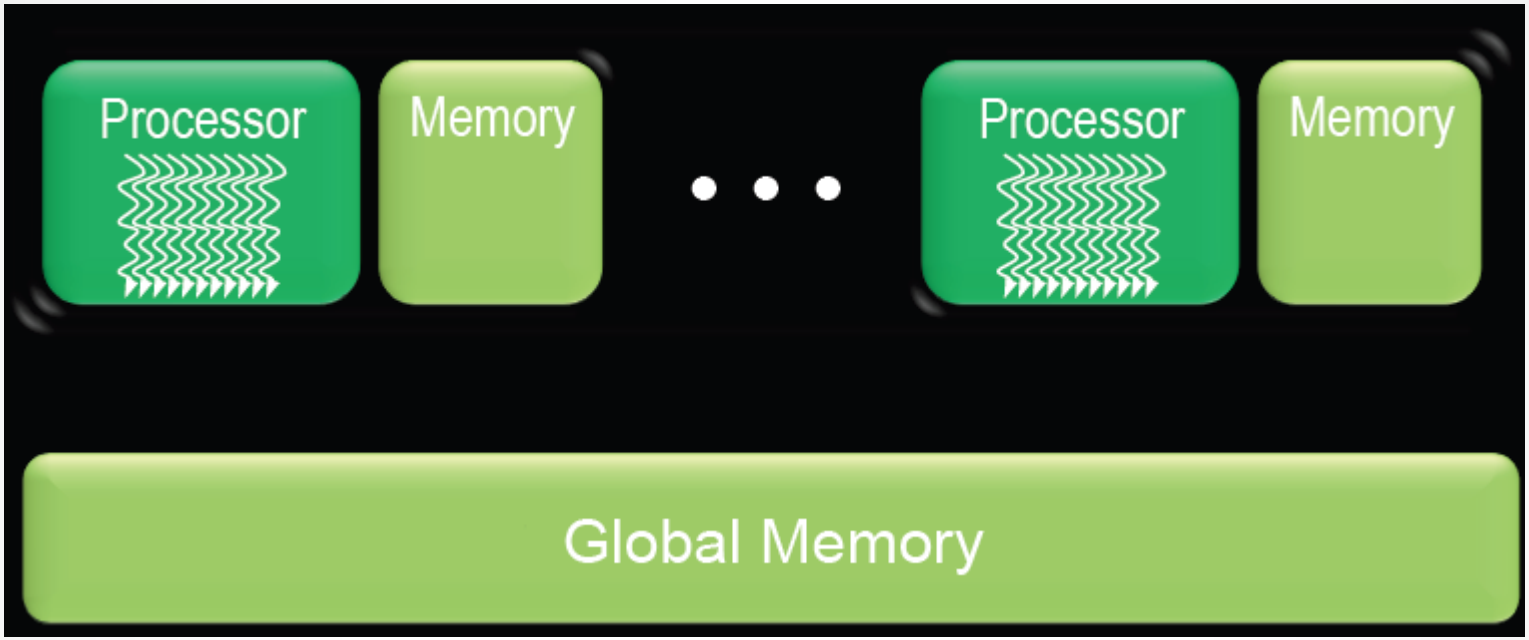
Manycore GPU

Generic Multicore Chip



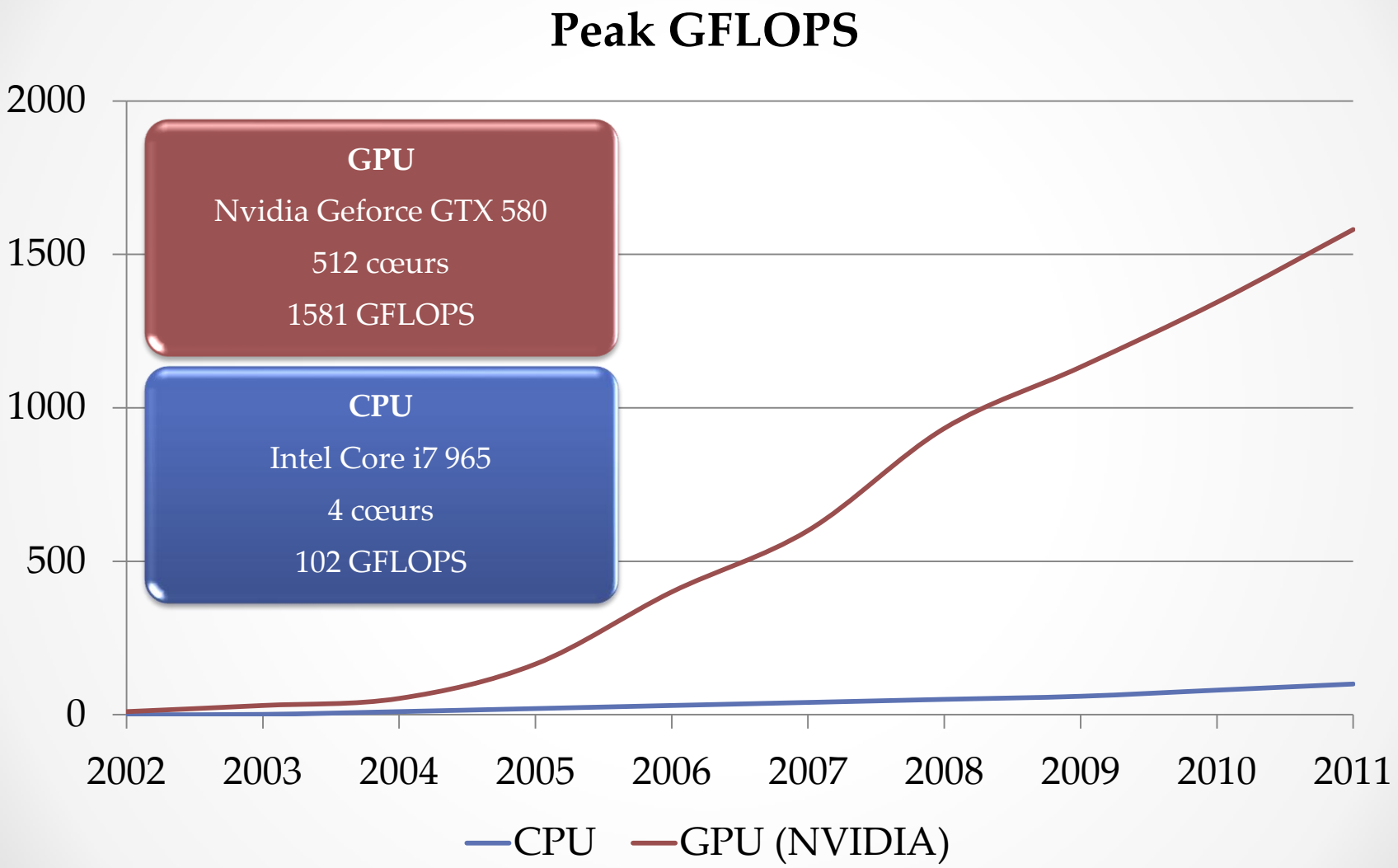
- Ensemble de processeurs chacun supportant 1 thread hardware
- Mémoire près des processeurs (cache, RAM, …)
- Mémoire globale partagée (DRAM Externe)

Generic Manycore Chip

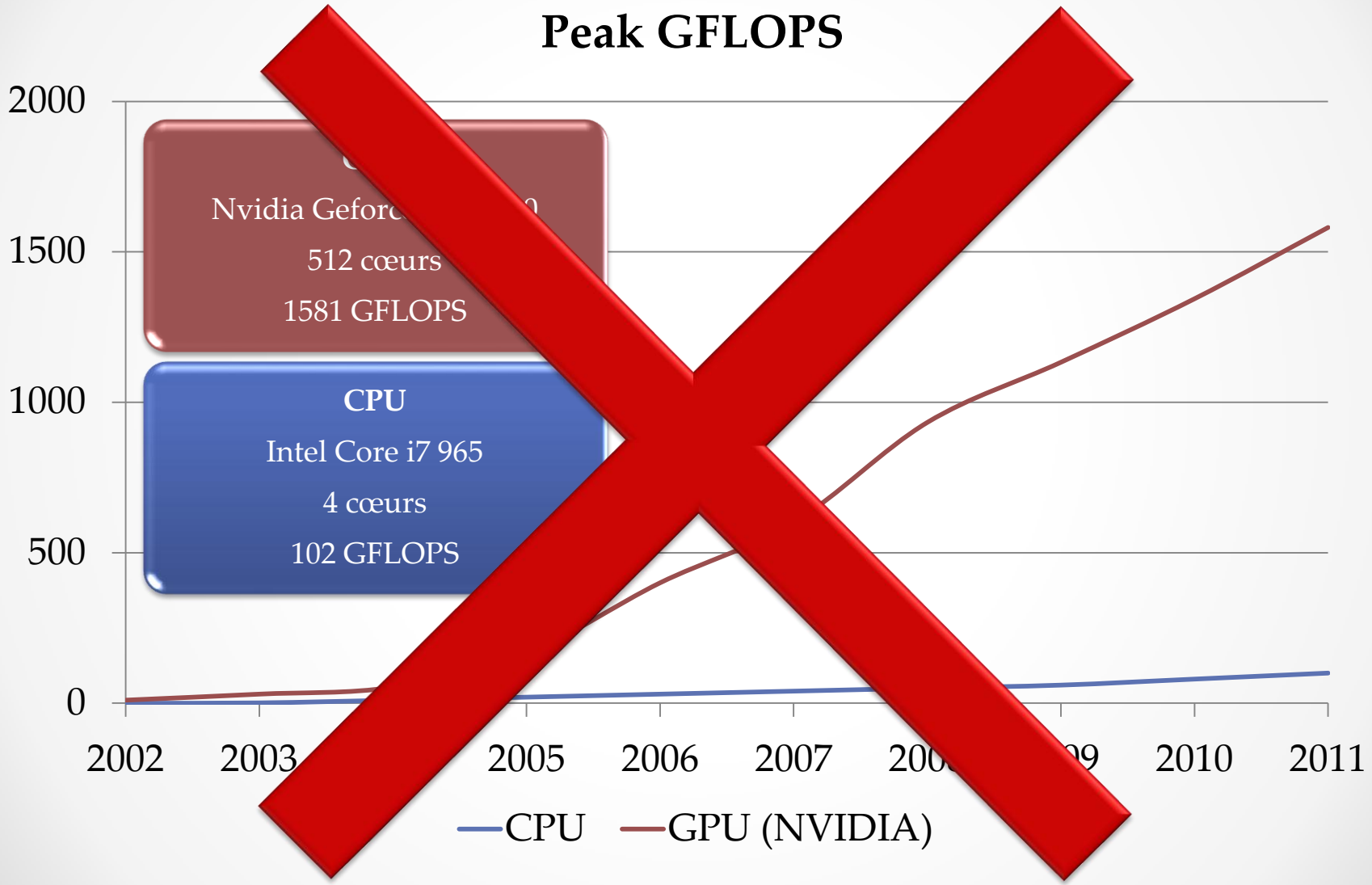


- Grand nombre de processeurs supportant chacun plusieurs threads hardware
- Mémoire près des processeurs (cache, RAM, …)
- Mémoire globale partagée (DRAM Externe)

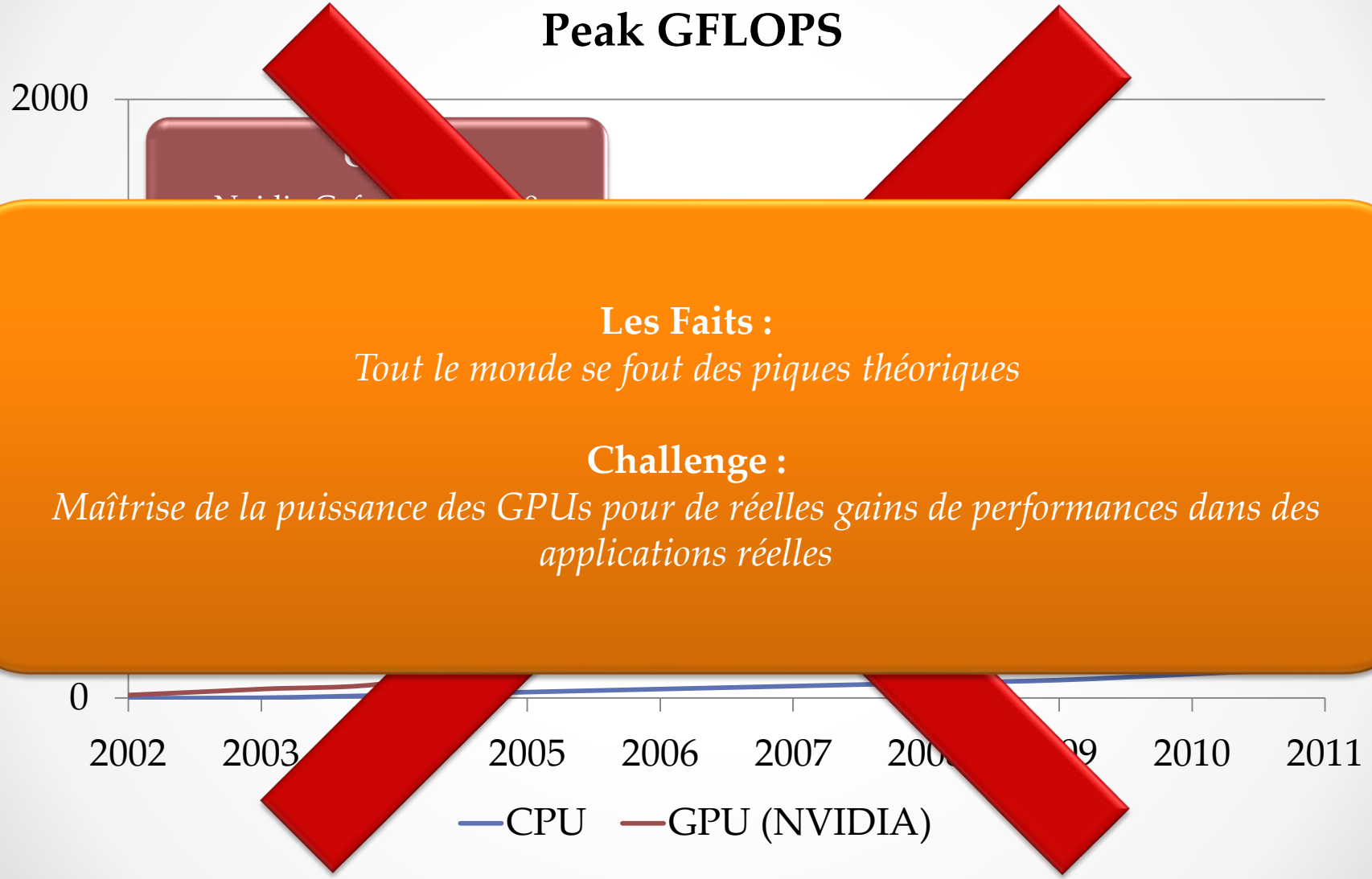
Motivation


Peak GFLOPS
2000
1500
1000
500
0
GPU
Nvidia Geforce GTX 580
512 cœurs
1581 GFLOPS
CPU
Intel Core i7 965
4 cœurs
102 GFLOPS
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
CPU
GPU (NVIDIA)

Motivation


Peak GFLOPS
2000
1500
1000
500
0
Nvidia Geforc
512 cœurs
1581 GFLOPS
CPU
Intel Core i7 965
4 cœurs
102 GFLOPS
2002
2003
2005
2006
2007
2010
2011
CPU
GPU (NVIDIA)

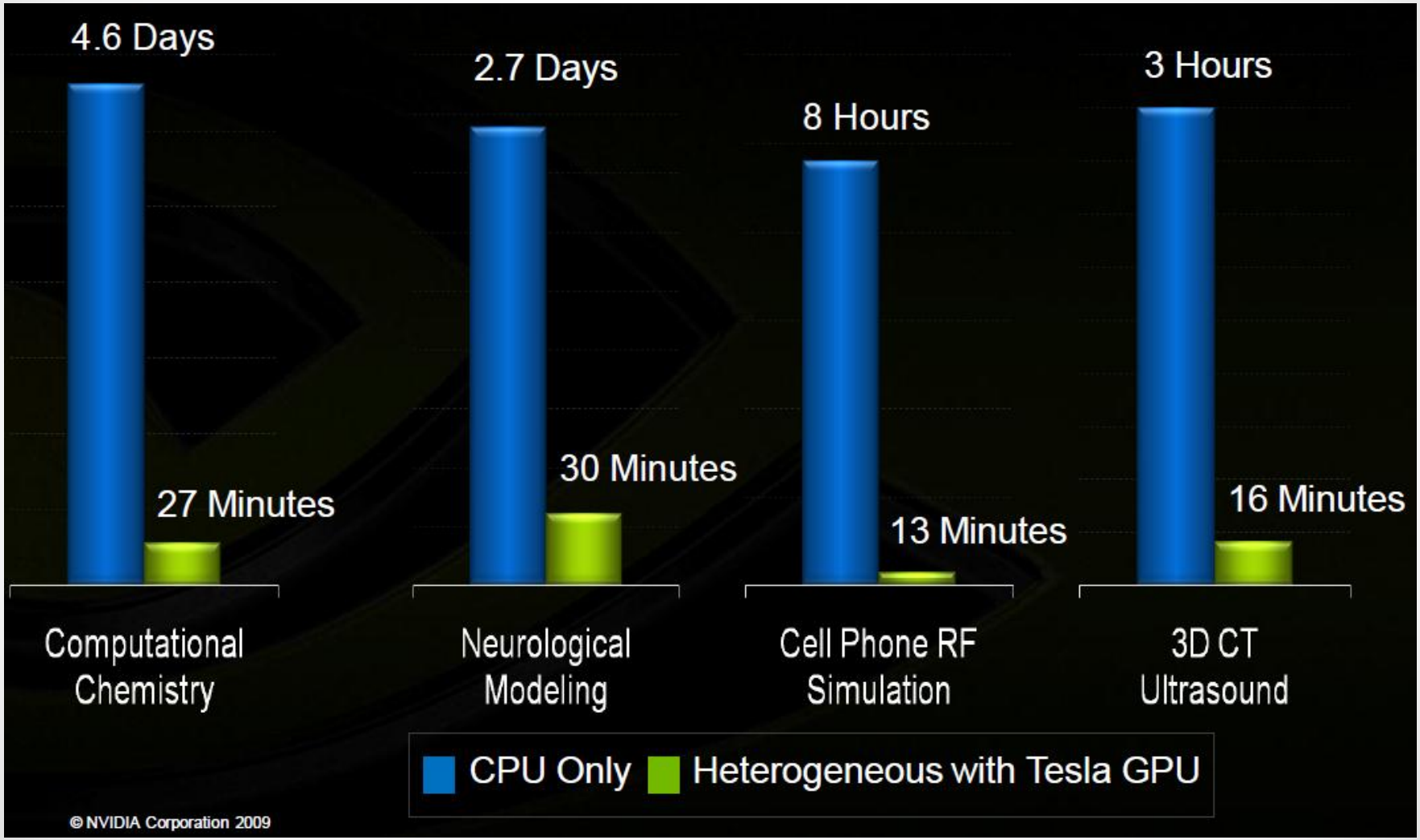
Motivation



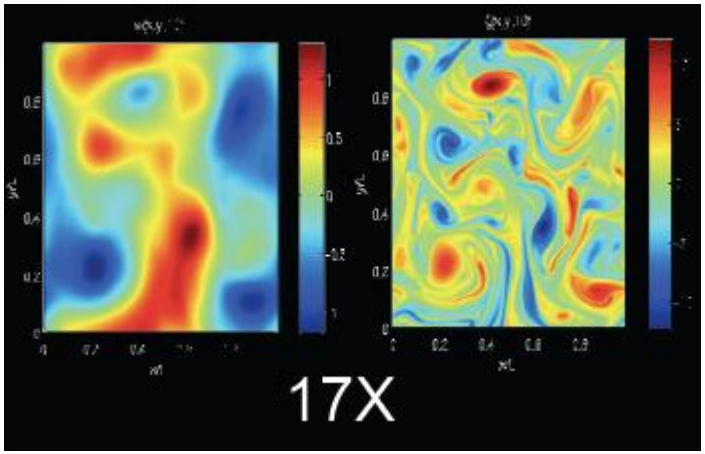
Les Faits :
Tout le monde se fout des piques théoriques

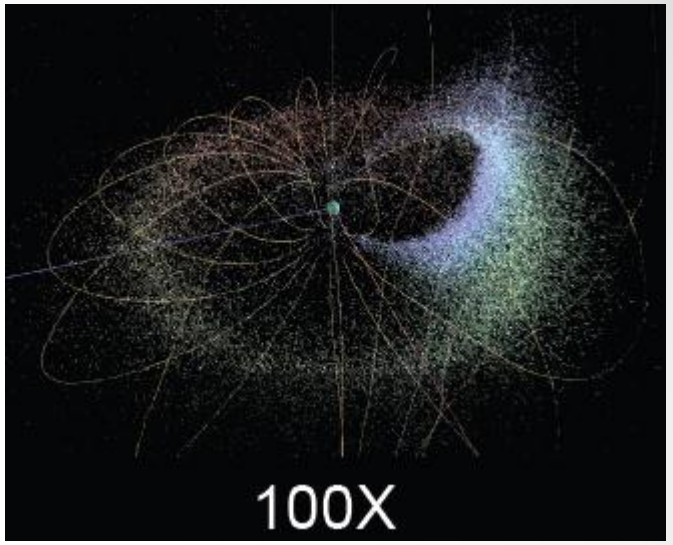
Challenge :
Maîtrise de la puissance des GPUs pour de réelles gains de performances dans des applications réelles

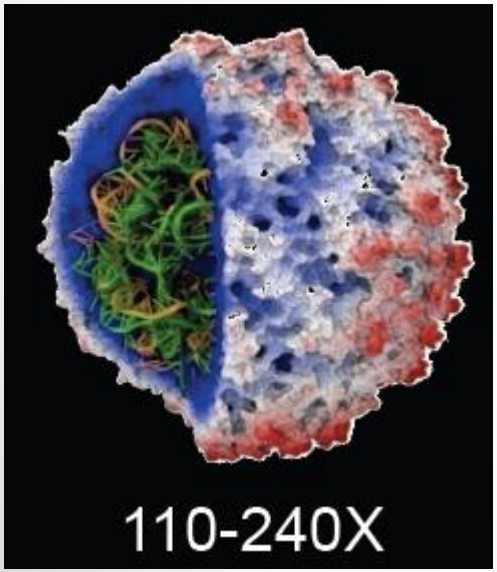
Motivation

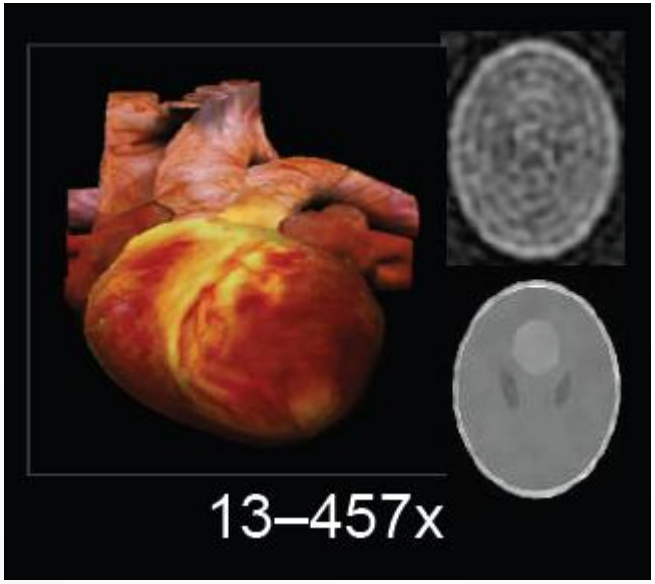

4.6 Days
27 Minutes
2.7 Days
30 Minutes
8 Hours
13 Minutes
3 Hours
16 Minutes
Computational Chemistry
Neurological Modeling
Cell Phone RF Simulation
3D CT Ultrasound
CPU Only
Heterogeneous with Tesla GPU
© NVIDIA Corporation 2009

Motivation









Disponibilité

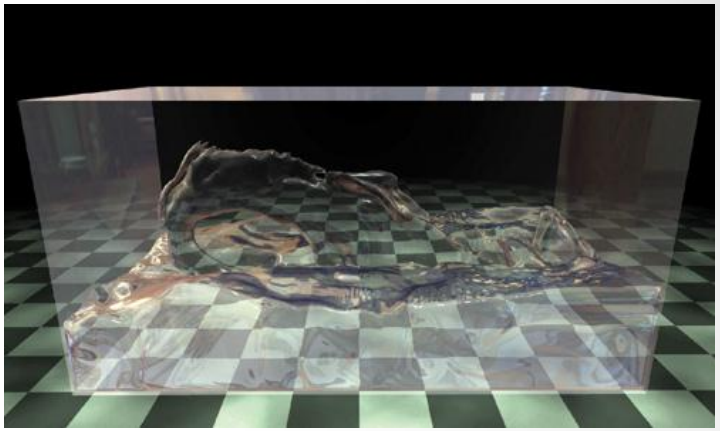
- Variété de produits : Du laptop au superordinateur

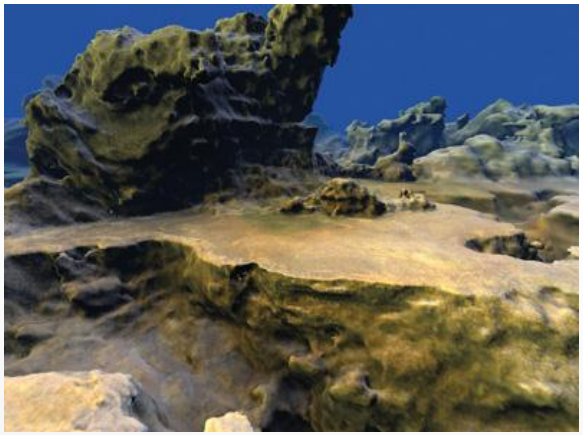
Histoire du GPGPU

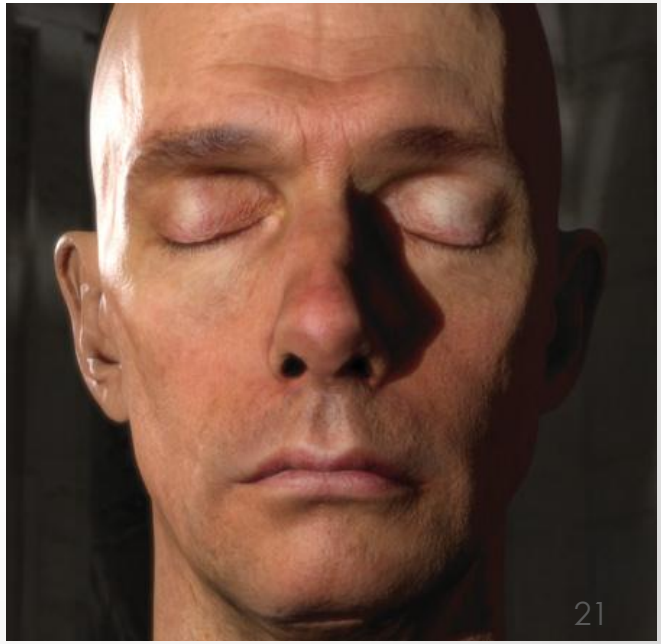
- Graphics in a Nutshell
- Le Pipeline Graphique
- Pourquoi les Gpus s'adaptent si bien ?
- Efficacité
- Les débuts du GPGPU

Graphics in a Nutshell

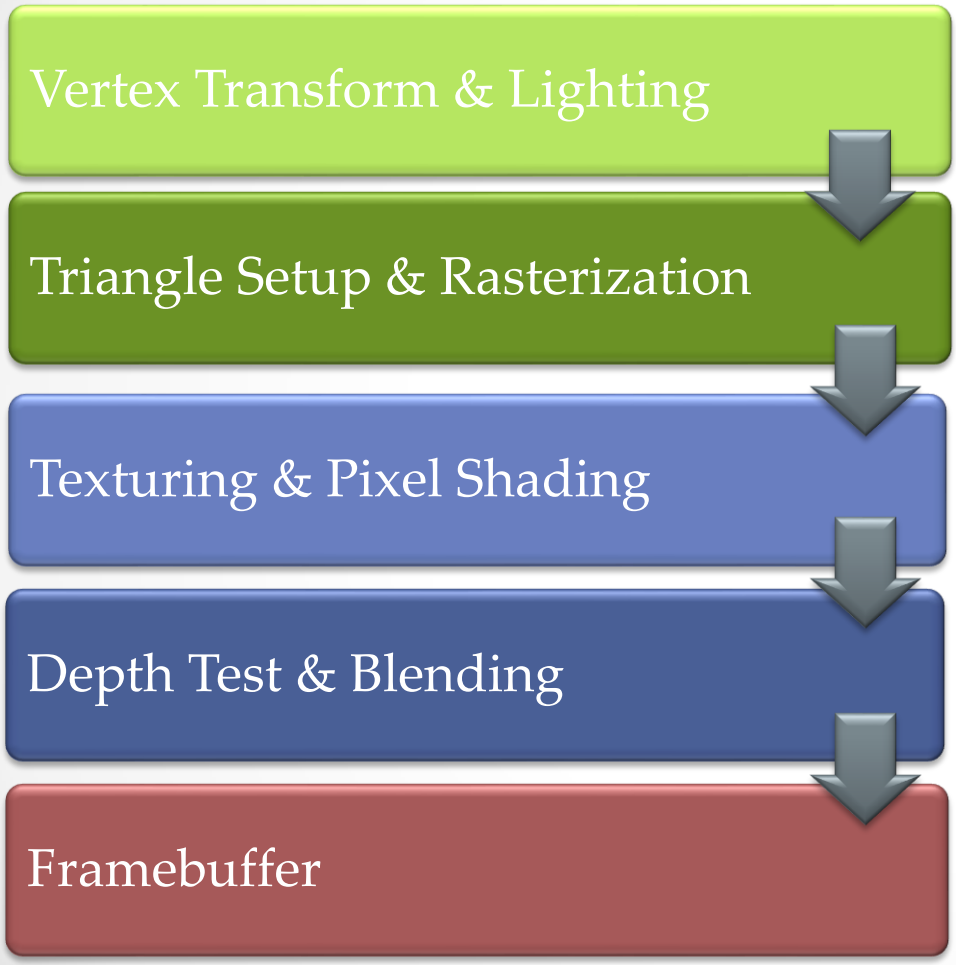
- Production de belles images
  - Formes complexes
  - Effets optiques complexes
  - Animations fluides
- Les produire rapidement
  - Invention de techniques astucieuses
  - Construction de matériel surpuissant



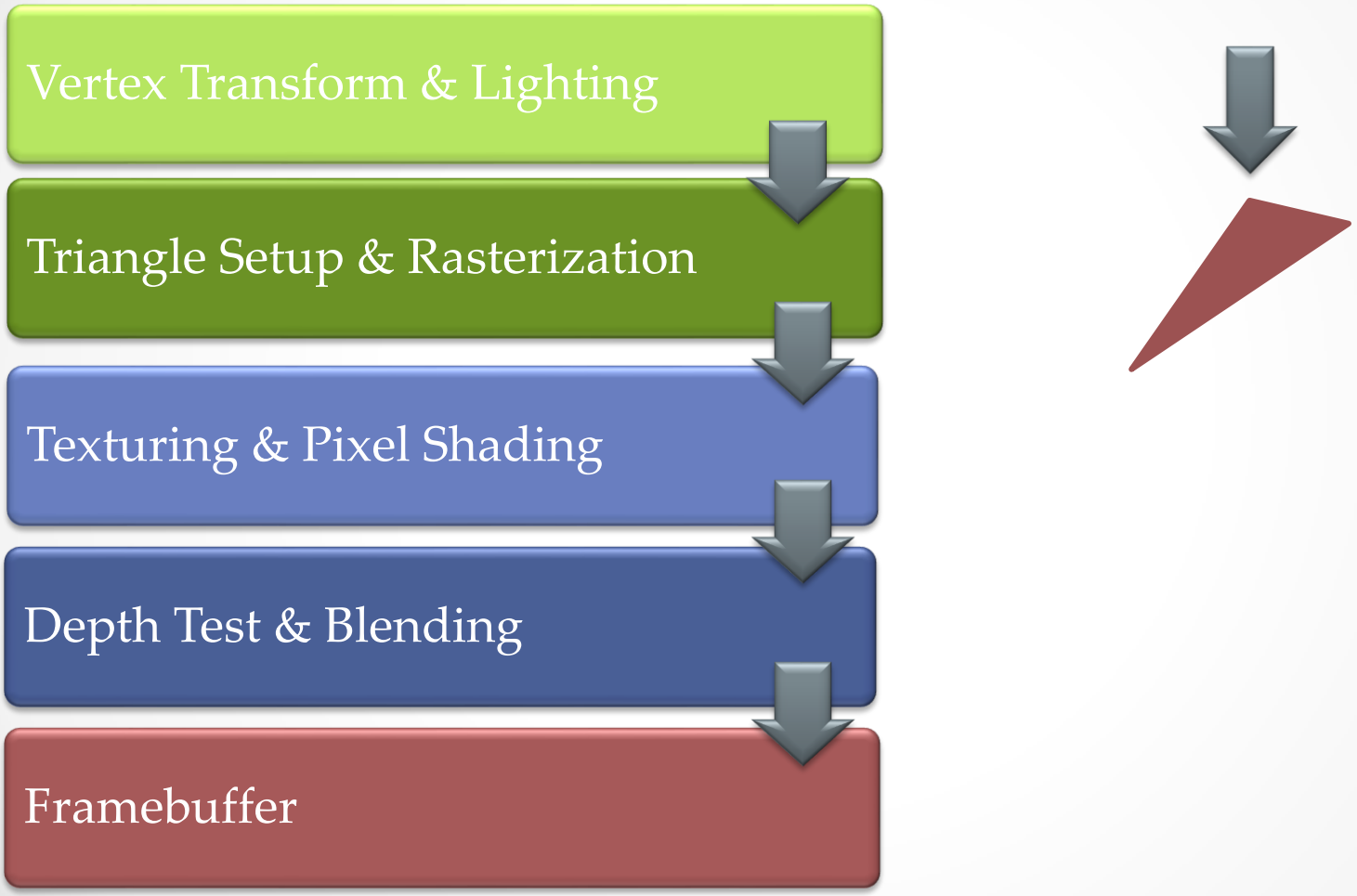




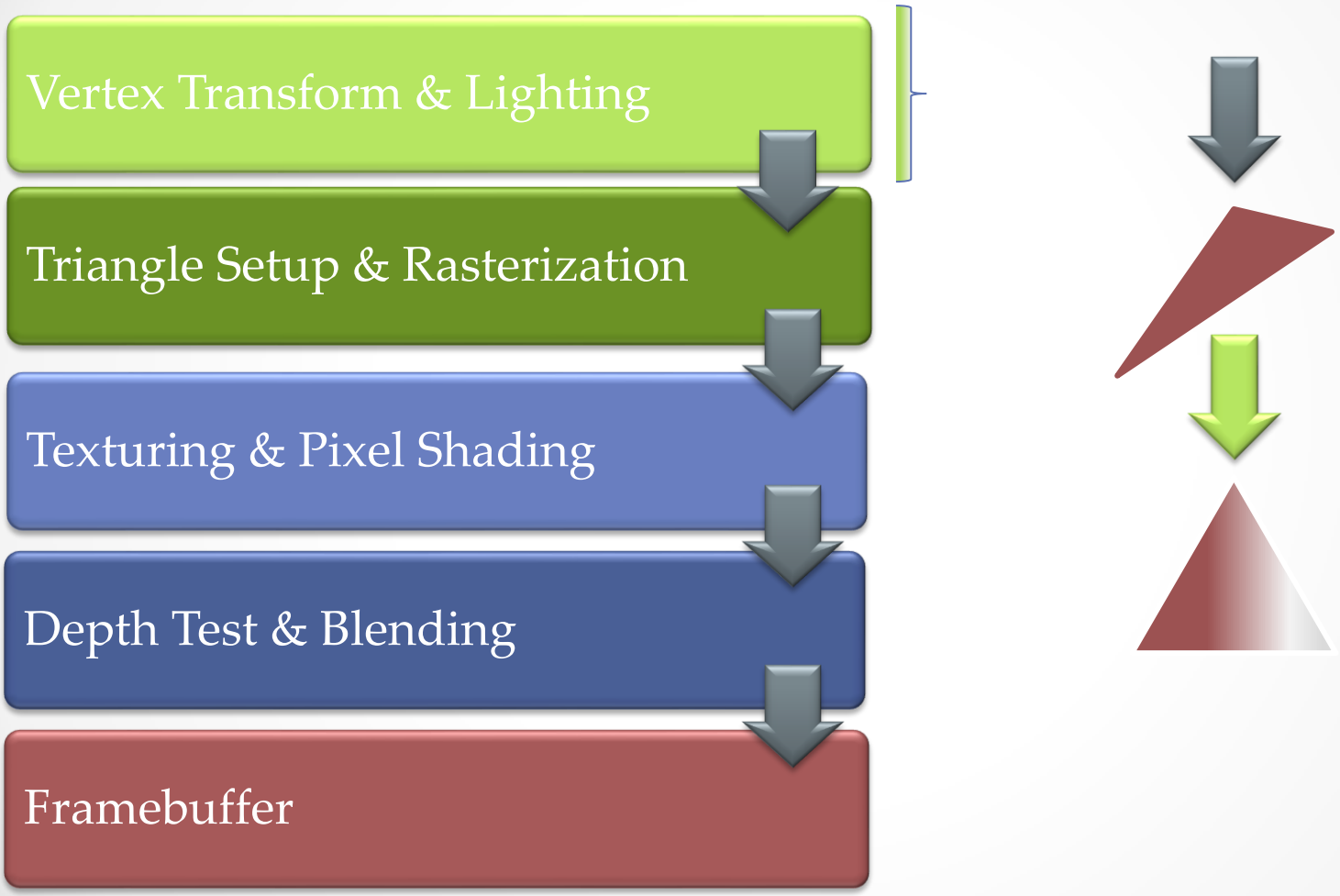
Le Pipeline Graphique



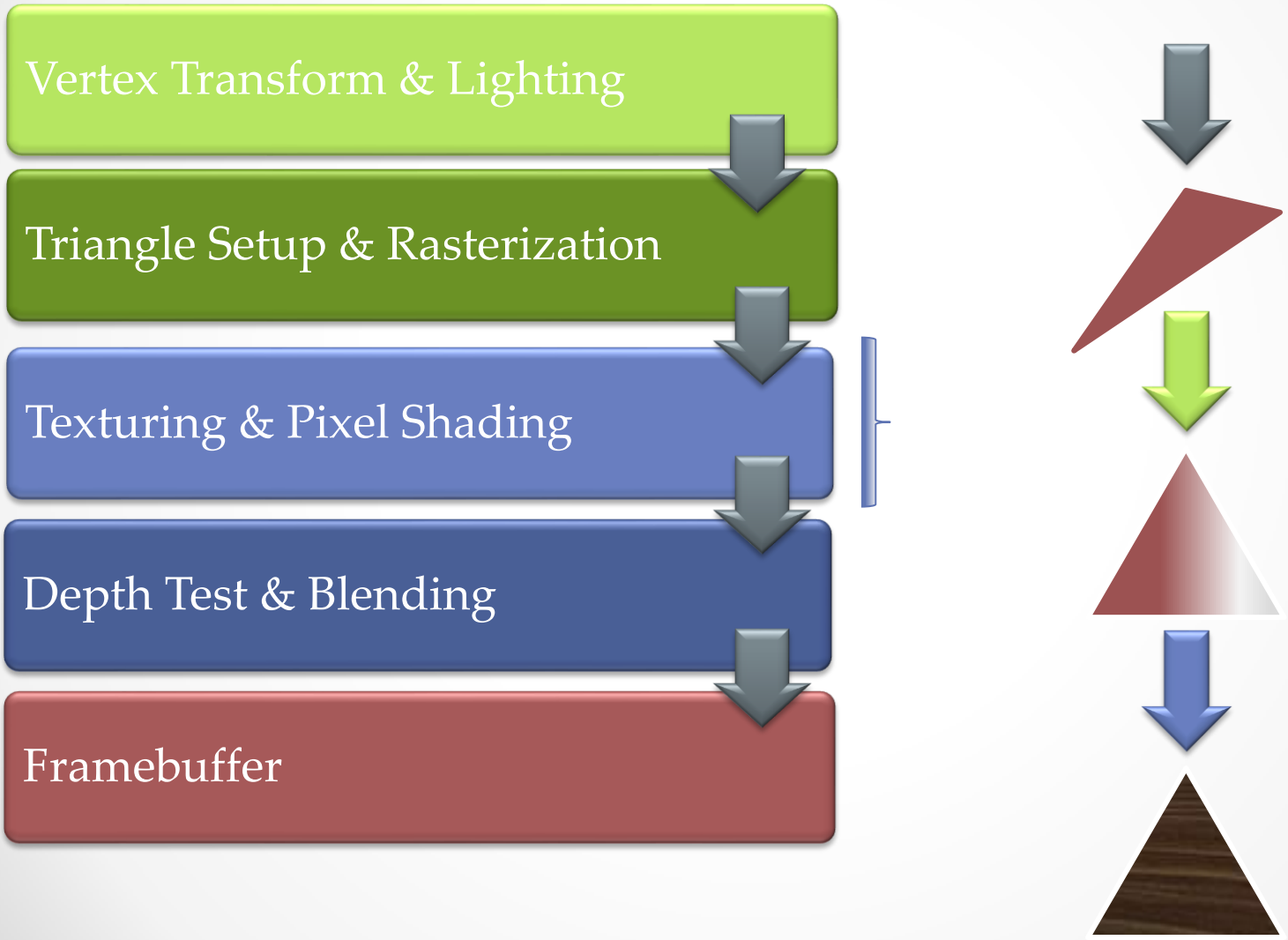
Le Pipeline Graphique



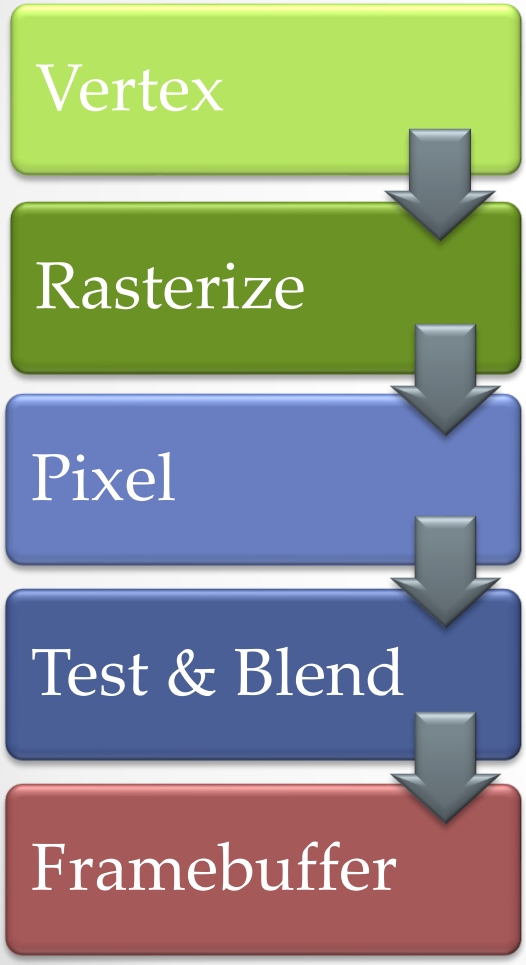
Le Pipeline Graphique



Le Pipeline Graphique

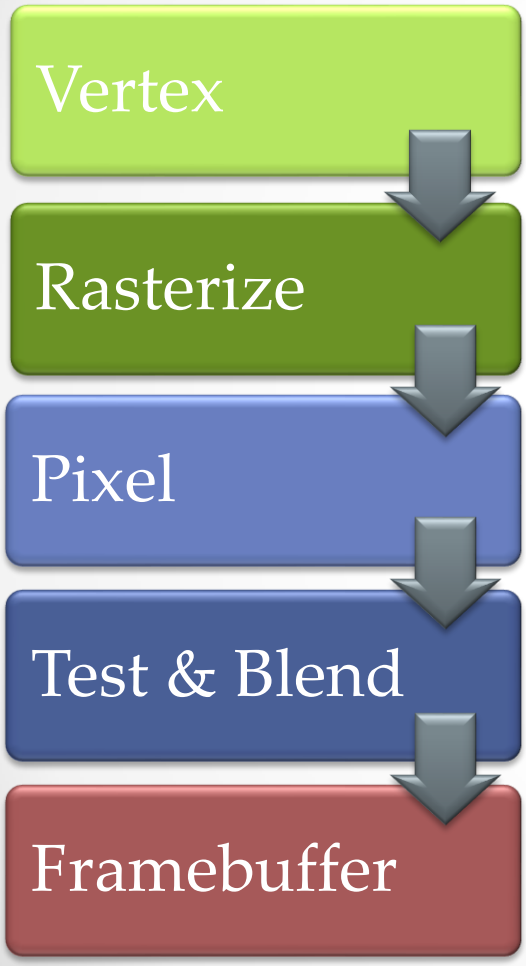


Le Pipeline Graphique



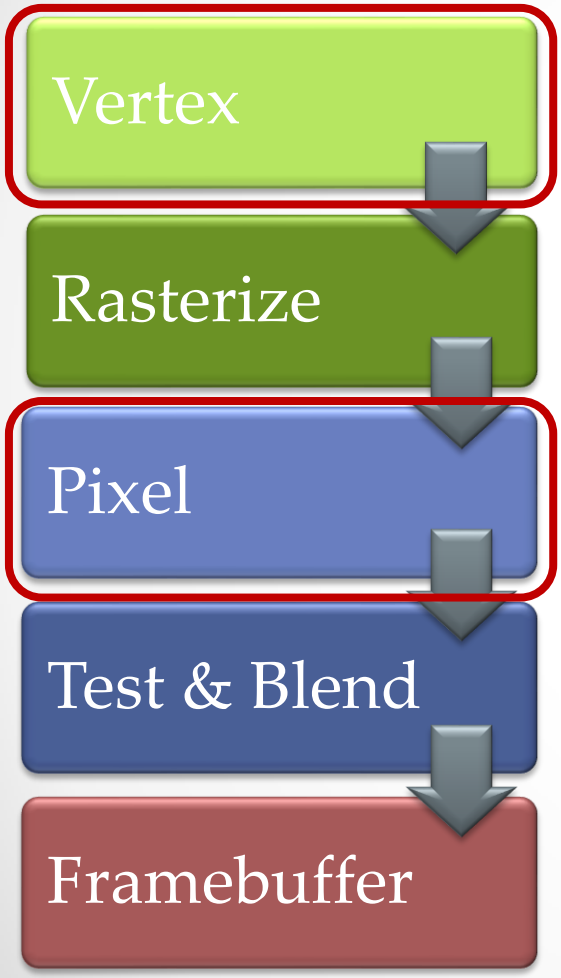
- Abstraction clé du graphisme temps-réel
- Hardware habituellement ressemblant à ceci
- Une puce par étape
- Les données traversent le pipeline

Le Pipeline Graphique



- Reste une abstraction clé
- Le hardware ressemble à ceci
- Traitement des vertex et pixel programmable
- Les GPU se focalisent de plus en plus sur l'exécution des shaders

Le Pipeline Graphique



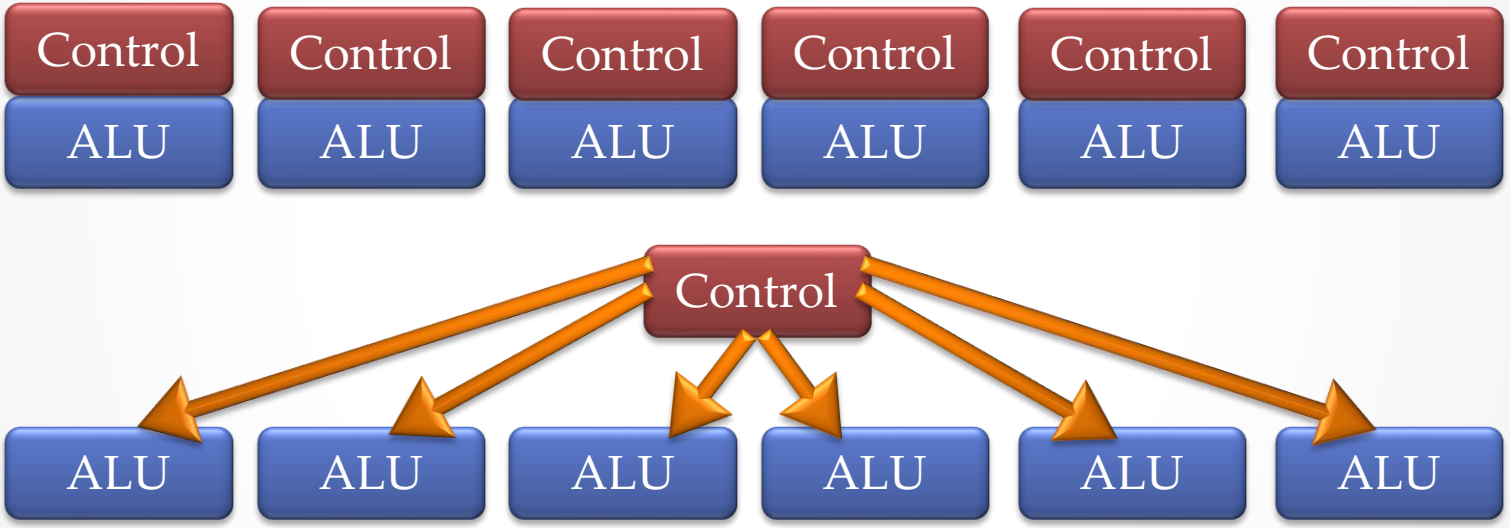
- Création de jeu d'instruction pour certaines étapes
- Instructions limitées

Pourquoi les gpus s'adaptent si bien ?

- Le type de charge de travail et le modèle de programmation fournissent énormément de parallélisme
- Les applications fournissent de large groupes de vertices en un coup
  - Les vertices peuvent être traité en parallèle
  - Application d'une même transformation à tout les vertices
- Les triangles contiennent de nombreux pixels
  - Les pixels d'un même triangle peuvent être traité en parallèle
  - Application du même shader à tout les pixels

Efficacité

- Le même code est exécuté pour un grand nombre de vertices/pixels



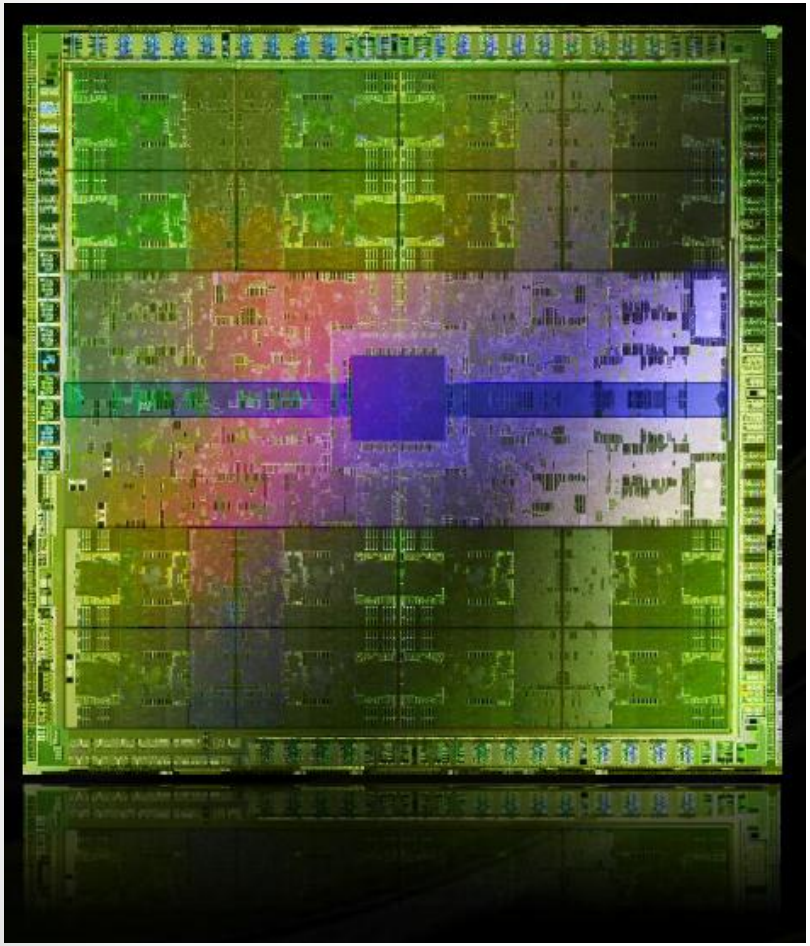
Les débuts du GPGPU

- Toutes ces performances attiraient les développeurs
- Pour utiliser les GPUs, ils devaient donc ré-exprimer leurs algorithmes par rapport à des calculs graphiques
- Travail très fastidieux → Facilité d'utilisation très limitée
- Très bon résultats obtenus !
- Tout cela à mené à CUDA

Architecture (Nvidia)

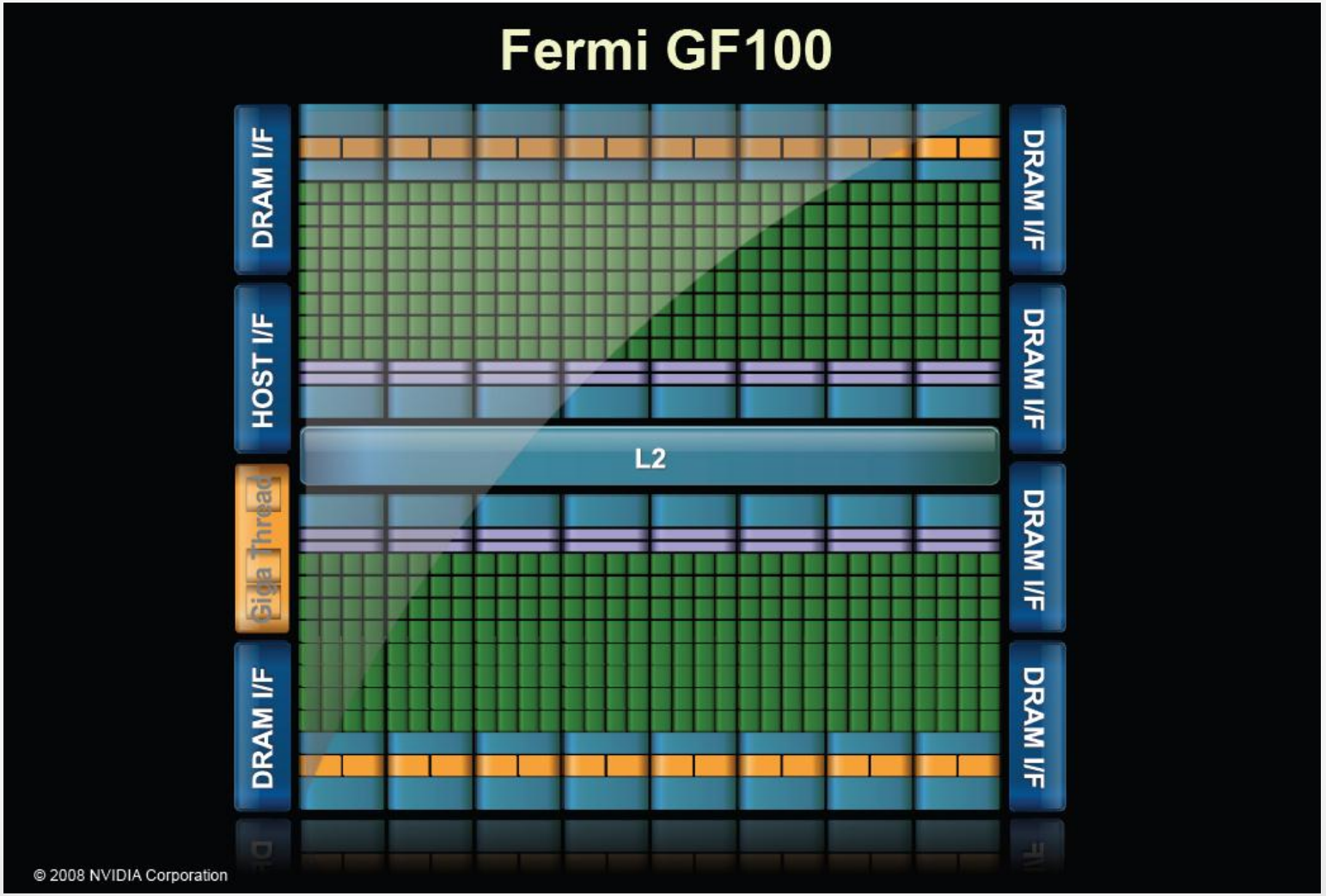
- NVIDIA GPU Architecture : Fermi GF100
- SM Multiprocessor
- Idées Architecturales clés

NVIDIA GPU Architecture



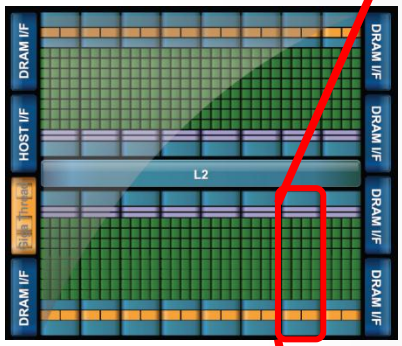
- GF100 ~ GTX 480
- 3 Milliards de transistors
- 512 cœurs CUDA
- ~ 2x bande passante mémoire
- L1 et L2 caches
- 8x peak fp64 performance
- ECC
- C++

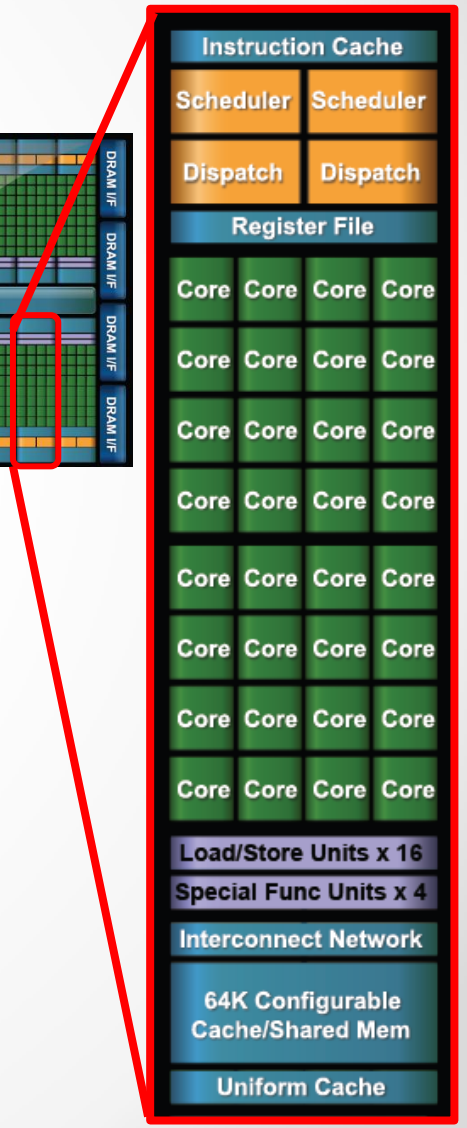
NVIDIA GPU Architecture



SM Multiprocessor

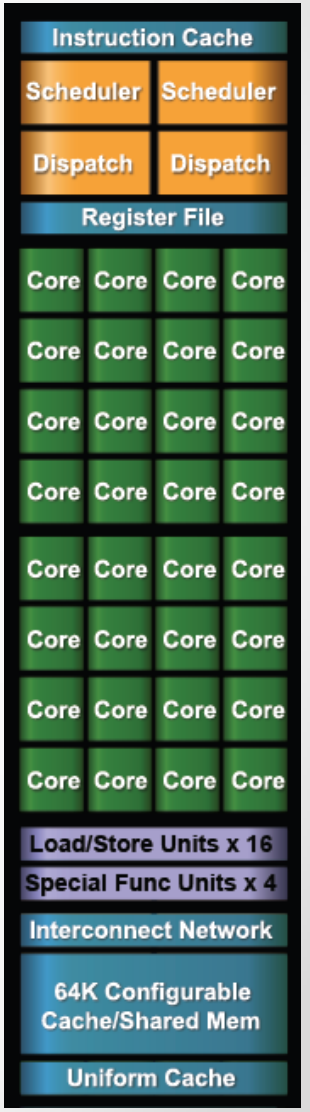
- 32 CUDA Cores par SM (512 total)
- 8x peak FP64 performance
  - 50% of peak FP32 performance
- Direct load/store to memory
  - Habituellement séquences linéaires de bytes
  - Haute bande passante (Centaines de GB/sec)
- 64KB de mémoire rapide partagée
  - Gestion Software ou Hardware
  - Partagé par les cœurs CUDA
  - Permettre la communication entre les threads





Idées Architecturales Clés

- Exécution SIMT (Single Instruction Multiple Thread)
  - Les threads s'exécutent dans des groupes de 32 appelé warps
  - Les threads dans un même warp partage l'unité d'instruction
  - L'hardware gèrent automatiquement les divergences
- Hardware multithreading
  - Allocation des ressources & Ordonnancement des threads
  - Latences ~ nombre de threads
- Les threads disposent de toutes les ressources nécessaires pour tourner
  - Aucun warp n'attend !
  - Switch de contexte est (basiquement) libre



CUDA

- Scalable Parallel Programming
- Abstractions Parallèles
- Hiérarchies de Threads Concurrents
- Modèle de Parallélisme en CUDA
- CUDA En Un Slide
- Programmation Hétérogène
- C pour CUDA
- Espaces Mémoires
- Mémoire GPU
- Copies De Données
- Exemple 1 : Addition de vecteurs
- Exemple 2 : Réduction parallèles

CUDA: Scalable parallel programming

- Enrichit le C/C++ avec des abstractions minimalistes
  - Laisser les programmeurs se concentrer sur les algorithmes parallèles
- Fournit une correspondance simple par rapport au matériel
  - S'adapte bien aux architectures GPU
  - S'adapte bien aussi aux CPUs multi-cœurs
- Centaines de cœurs vs. Milliers de threads parallèles
  - Les threads GPU sont léger – Create/Switch gratuit
  - Les GPU ont besoin de milliers de threads pour être pleinement utilisés

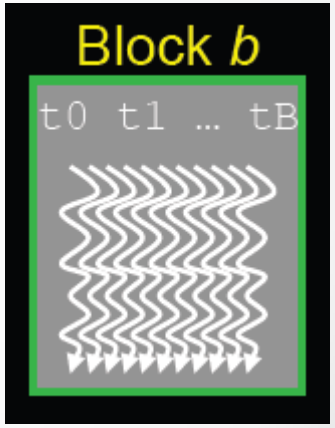
Abstractions Parallèles

- Hiérarchie de threads concurrents
- Synchronisation légère
- Modèle de mémoire partagée pour la coopération entre threads

Hiérarchies de Threads Concurrents

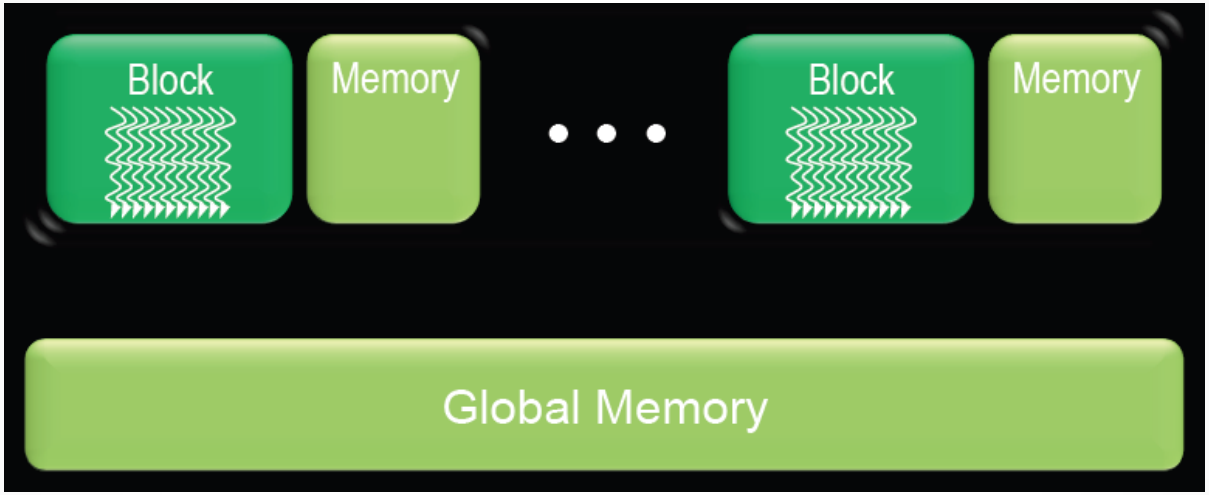
- Parallel kernels : Composé de nombreux threads
  - Tout les threads exécutent le même programme séquentielle

- Threads blocks : Les threads sont regroupés par bloques
  - Les threads dans un même bloque peuvent coopérer :
    - Synchronisation de leurs exécution
    - Communication via la mémoire partagée du bloque



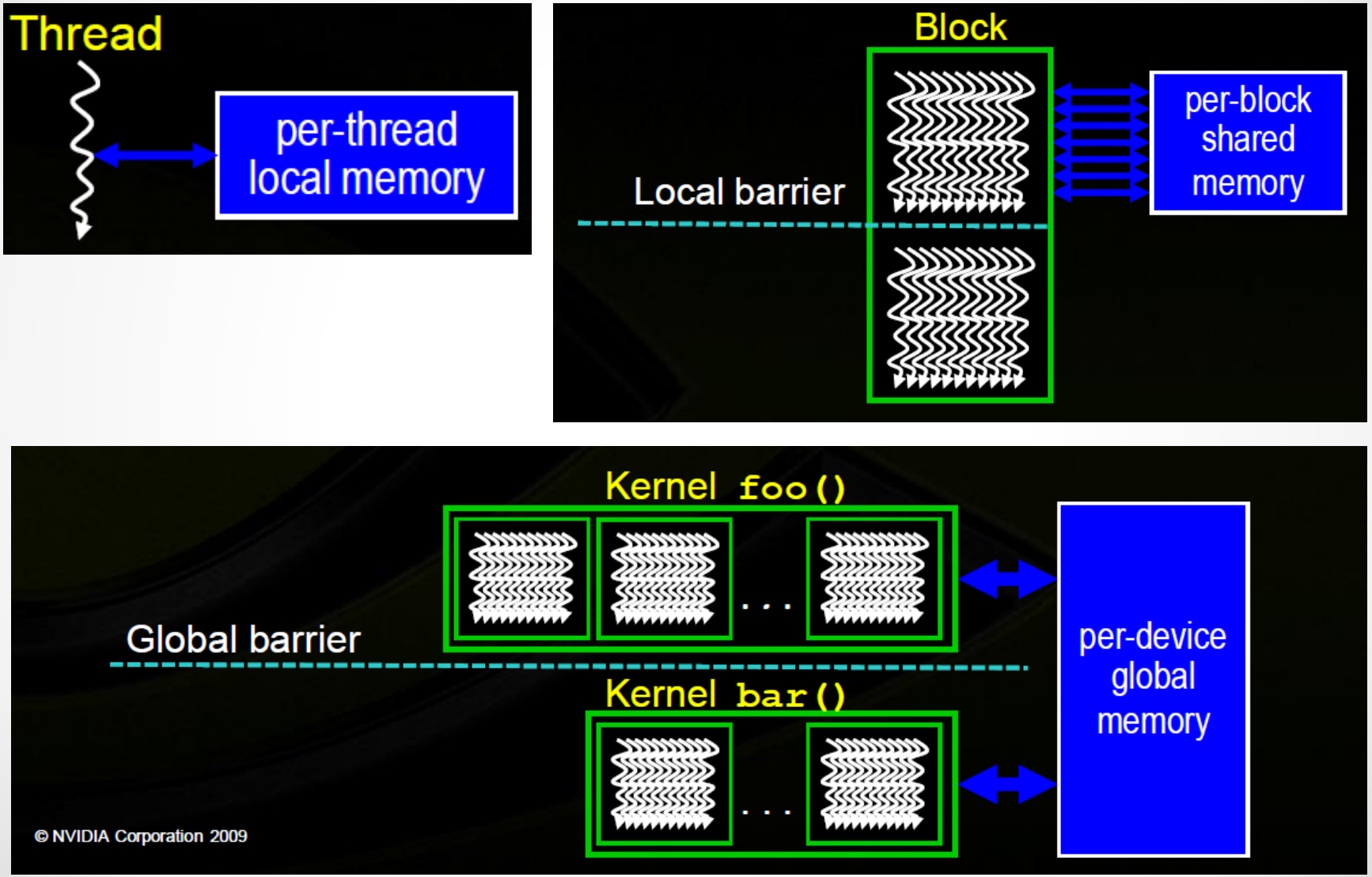
- Threads et bloques ont des Ids uniques

Modèle de Parallélisme en CUDA



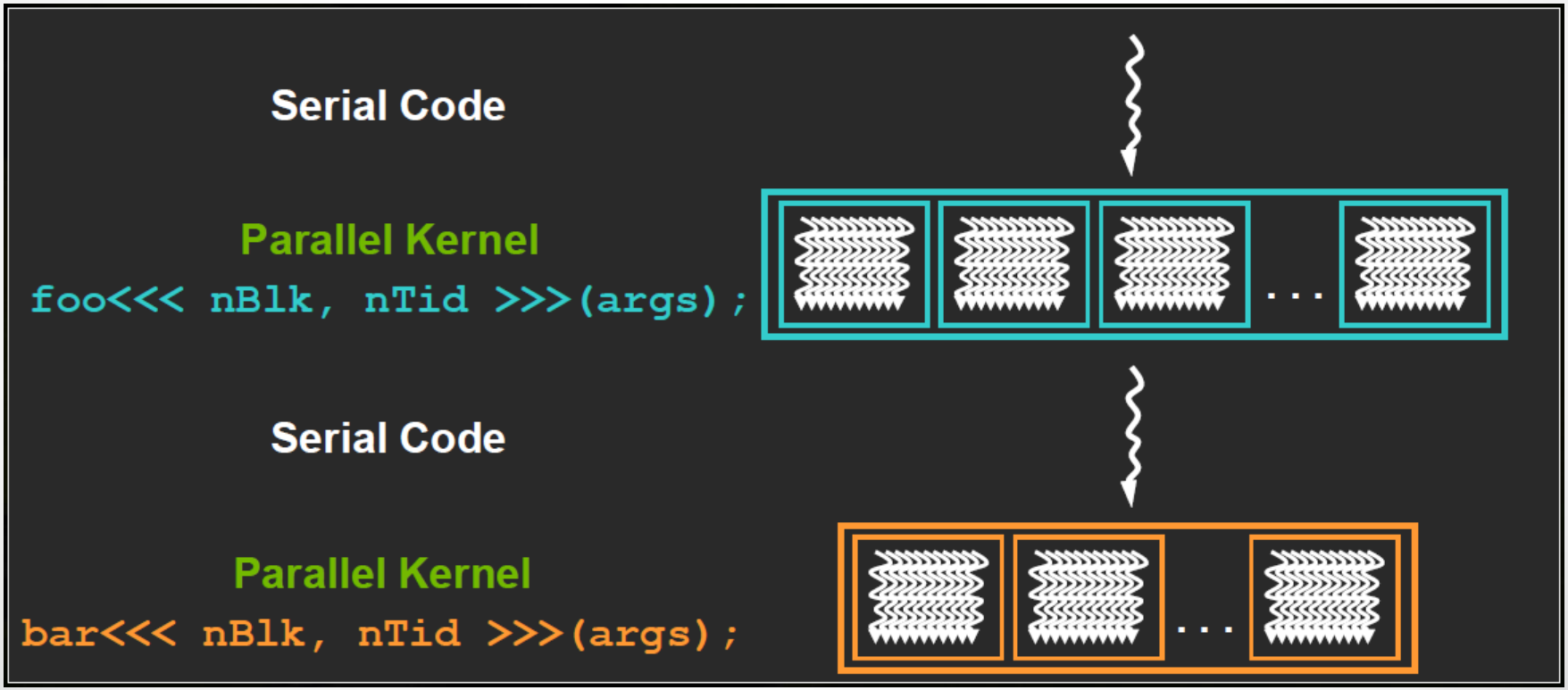
- CUDA virtualise le matériel physique
  - Un thread est un processeur scalaire virtualisé
  - Un bloque est un multiprocesseur virtualisé
- Ordonnancement sur le matériel physique sans préemption
  - Threads/Blocks chargé et exécuté jusqu'à exécution complète
  - Les bloques doivent être indépendants

CUDA En Un Slide



Programmation Hétérogène

- Utilisation du bon processeur pour les bonnes tâches :



C pour CUDA

- Philosophie: *Fournir un ensemble minimal d'extensions nécessaires*
- Qualificateurs de fonctions :

```
__global__ void  my_kernel()       {}
__device__ float my_device_func()  {}
```

- Qualificateurs de variables :

```
__constant__ float my_constant_array[32];
__shared__   float my_shared_array[32];
```

- Configuration d'exécution :

```
dim3 grid_dim(100, 50);   // 5000 thread blocks
dim3 block_dim(4, 8, 8);  // 256 threads per block
my_kernel() <<< grid_dim, block_dim >>> (…); // Launch kernel
```

- Variables et fonctions « Built-in » :

```
dim3 gridDim;     // Grid dimension
dim3 blockDim;    // Block dimension
dim3 blockIdx;    // Block index
dim3 threadIdx;   // Thread index
void __syncthreads();    // Thread synchronization
```

Espaces Mémoires

- CPU & GPU ont des espaces mémoires différents
  - Les données transitent par le bus PCIe
  - Utilisation de fonctions pour allouer/initialiser/copier la mémoire sur le GPU
    - Très similaire aux fonctions C correspondante
- Les pointeurs sont juste des adresses
  - Impossibilité de différencié une adresse CPU d'une adresse GPU
  - Attention donc lors du déférencement:
    - Pointeur CPU en pointeur GPU → Crash
    - & Vice Versa

Mémoire GPU

- Le host (CPU) gère la mémoire device (GPU) :
  - `cudaMalloc(void** pointer, size_t nbytes);`
  - `cudaMemset(void* pointer, int value, size_t nbytes);`
  - `cudaFree(void* pointer);`
- Exemple :

```
int n = 1024;
int nbytes = 1024 * sizeof(int);
int *d_a = 0;
cudaMalloc((void**) &d_a, nbytes);
cudaMemset(d_a, 0, nbytes);
cudaFree(d_a);
```

Copies De Données

- `cudaMemcpy(void* dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
  - Retour une fois la copie complète
  - Thread CPU bloqué jusqu'à ce que tout les bytes soient copiés
  - Copie ne commence qu'une fois touts les précédents appels CUDA sont complétés

- `enum cudaMemcpyKind`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`

- Remarque : Copies non-bloquantes aussi disponibles !

Exemple 1 : Addition de vecteurs

```
// Calcul de la somme de vecteurs c = a + b
// Chaque thread exécute une addition sur une paire d'éléments
__global__ void  vector_add(float* A, float* B, float* C)
{
        int i = threadIdx.x + blockDim.x * blockIdx.x;
        C[i] = A[i] + B[i];
}
```

Device Code (GPU)

```
int main()
{
        // Code d'initialisation
        …
        // Chargement de N/256 bloques de 256 threads chacun
        vector_add<< N/256, 256 >>(d_A, d_B, d_C);
}
```

Host Code (CPU)

Exemple 1 : Host Code

```
// Allouer et initialiser host memory (CPU)
float *h_A = …, *h_B = …;

// Allouer et initialiser device memory (GPU)
float *d_A, *d_B, *d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));

// Copie de la host memory à la device memory
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice);

// Exécution d'une grille de N/256 bloques de 256 threads chacun
vector_add<< N/256, 256 >>(d_A, d_B, d_C);
```

Exemple 1 : Host Code (suite)

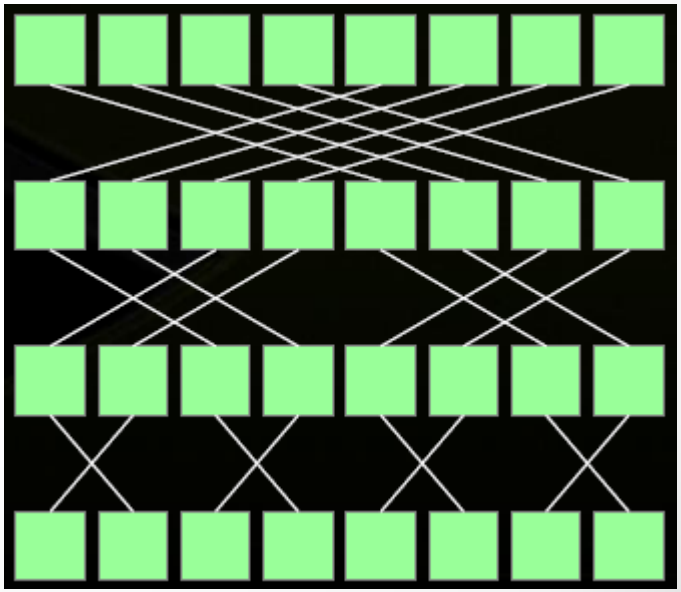
```
// Exécution d'une grille de N/256 bloques de 256 threads chacun
vector_add<< N/256, 256 >>(d_A, d_B, d_C);

// Copie du résultat dans la host memory (CPU)
cudaMemcpy(h_A, d_A, N * sizeof(float), cudaMemcpyDeviceToHost);
cudaMemcpy(h_B, d_B, N * sizeof(float), cudaMemcpyDeviceToHost);

// Faire quelque chose avec le résultat…

// Libérer la device memory (GPU)
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Exemple 2 : Réduction Parallèle

- Somme des éléments d'un vecteur avec 1 thread :

```
int sum = 0;
for(int i = 0; i < N; ++i)
    sum += x[i];
```

- La réduction parallèle se construit comme un arbre d'addition :
  - Chaque thread retient un élément
  - Sommes partielles pas à pas
  - *N* threads nécessitent *log N* étapes
  - Approche possible : Butterfly pattern

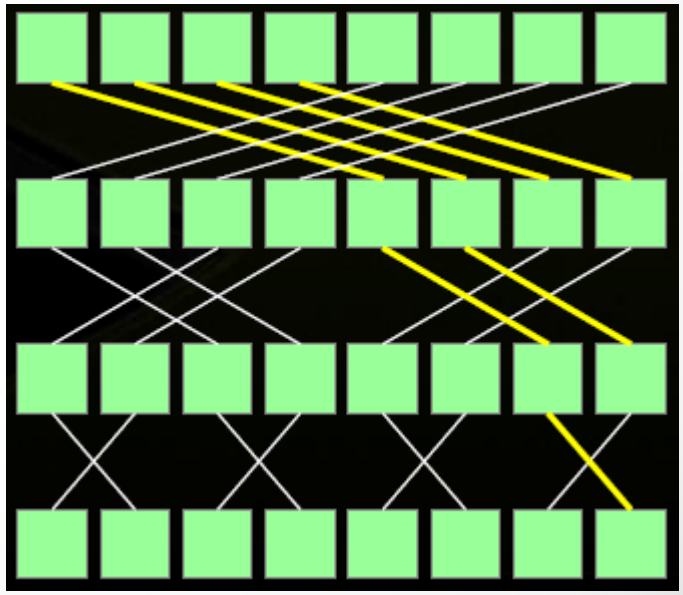


Exemple 2 : Réduction Parallèle

- Somme des éléments d'un vecteur avec 1 thread :

```
int sum = 0;
for(int i = 0; i < N; ++i)
    sum += x[i];
```

- La réduction parallèle se construit comme un arbre d'addition :
  - Chaque thread retient un élément
  - Sommes partielles pas à pas
  - *N* threads nécessitent *log N* étapes
  - Approche possible : Butterfly pattern



Exemple 2 : Réduction Parallèle pour un bloque

```
// INPUT : Thread i holds value x_i
int i = threadIdx.x;
__global__ int sum[blocksize];

// Un thread par élément
sum[i] = x_i;
__syncthreads();

for(int bit = blocksize/2; bit > 0; bit /= 2)
{
        int t = sum[i] + sum[i^bit];
        __syncthreads();
        sum[i] = t;
        __syncthreads();
}

// OUTPUT : Chaque thread contient maintenant la somme dans sum[i]
```

Indépendance des bloques

- Tout entrelacement de bloques doit être valide
  - Exécution sans préemption
  - Exécution dans n'importe quel ordre
  - Exécution concurrente ou séquentielle
- L'indépendance permet :
  - L'évolutivité !
  - L'extensibilité à grande échelle

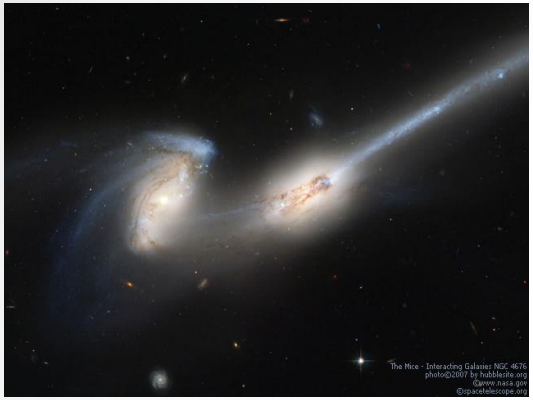
Remarques

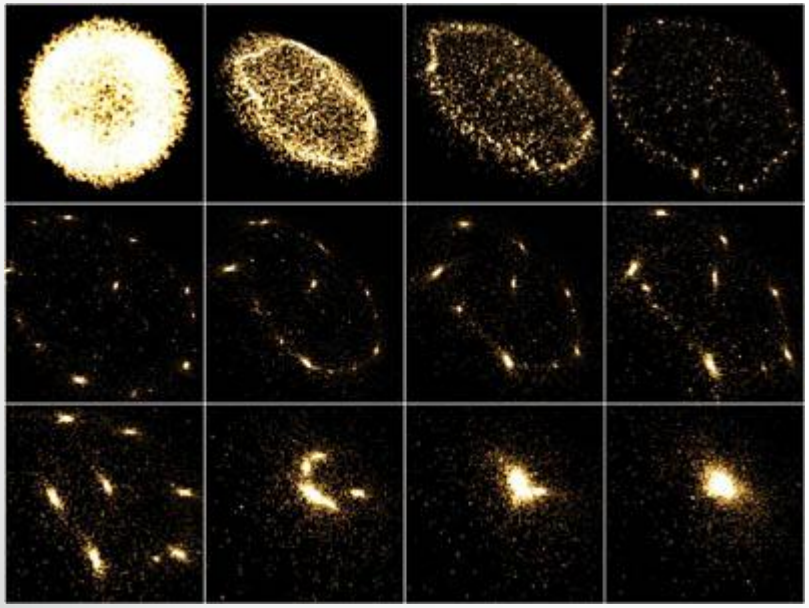
- C/C++ mais avec restrictions :
  - Accès à la mémoire GPU uniquement
  - Pas de nombre variable d'arguments
  - Pas de variable statique
  - Pas de récursion
  - Pas de polymorphisme dynamique

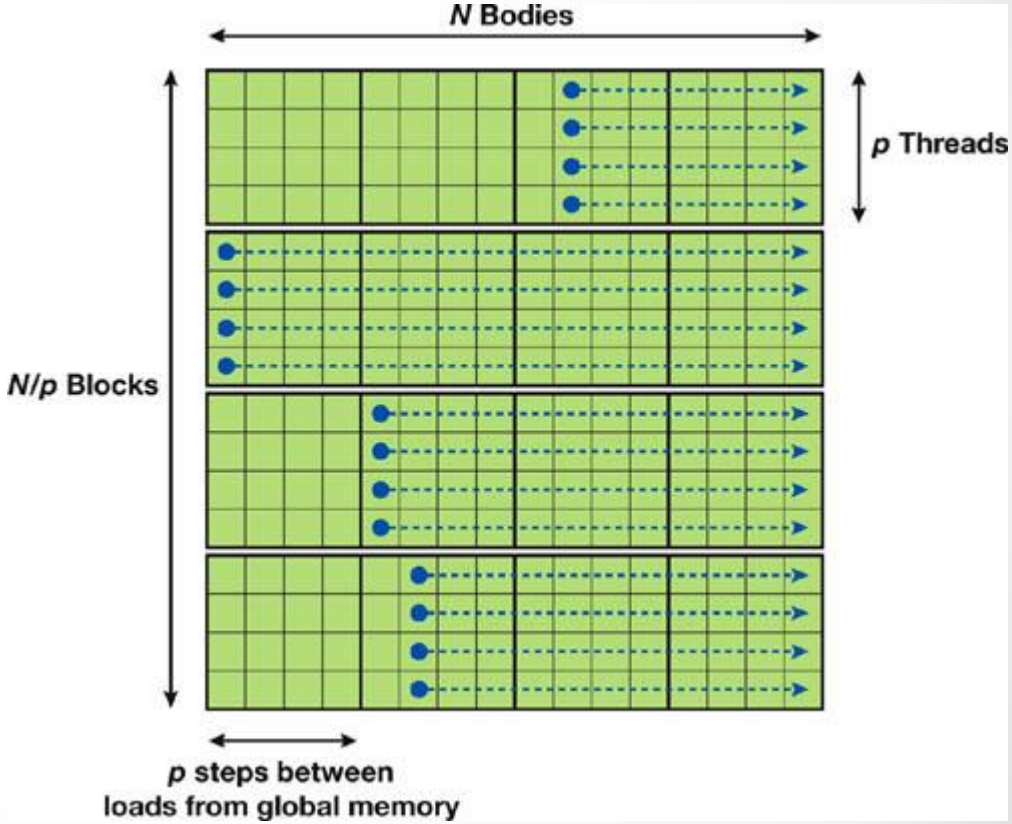
Applications

- N-Body Dynamics
- Interactive Ray-Tracing
- Neural Networks
- PhysX
- Optix
- Medical Imaging
- Computational Fluid Dynamics
- Environmental Science

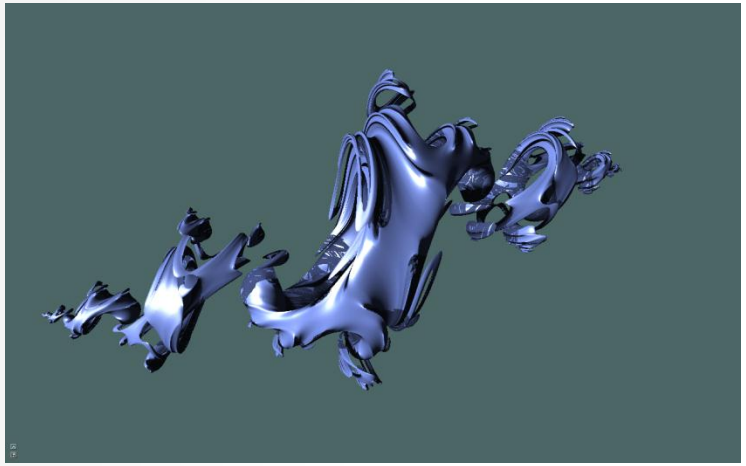
Application : N-Body Dynamics

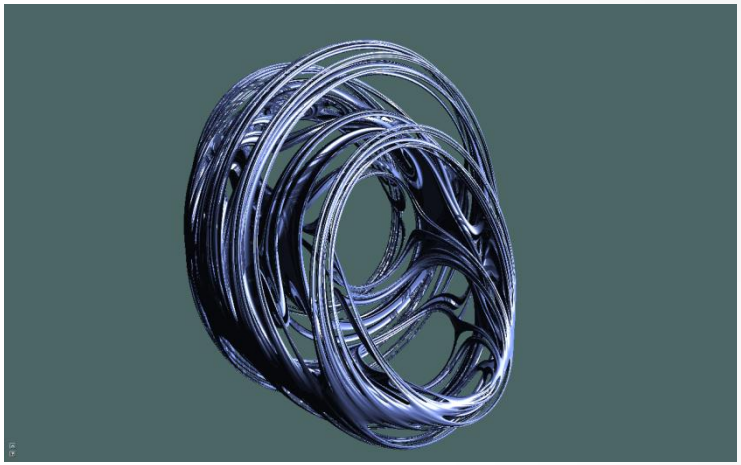


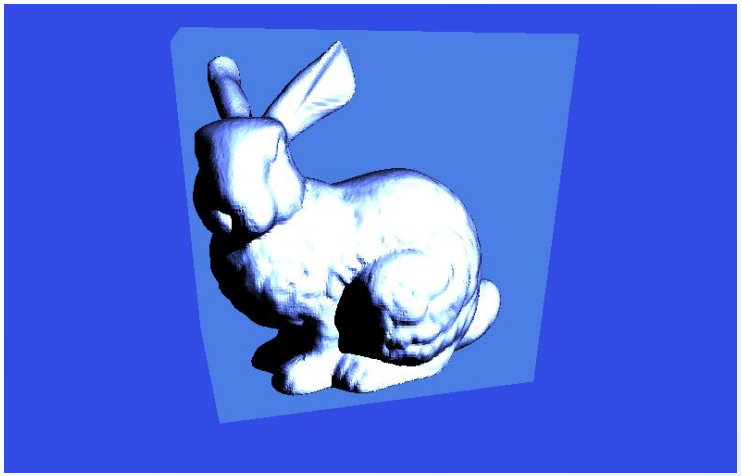




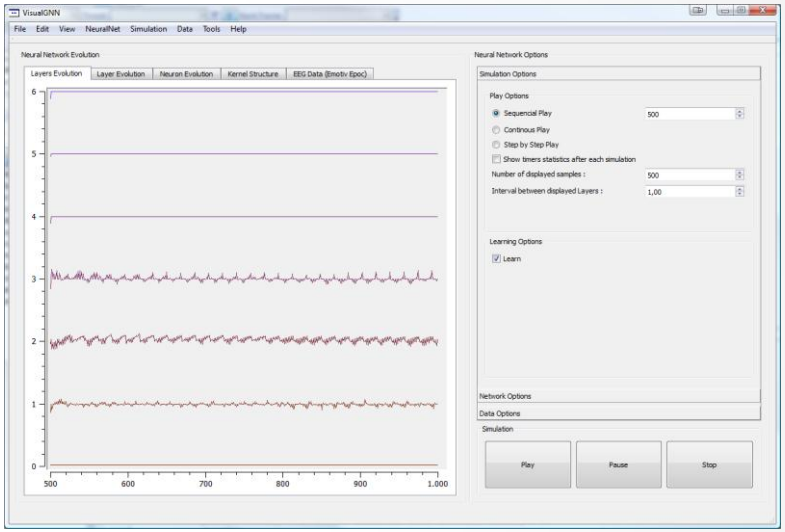
Application : Interactive Ray-Tracing







Application : Neural Networks

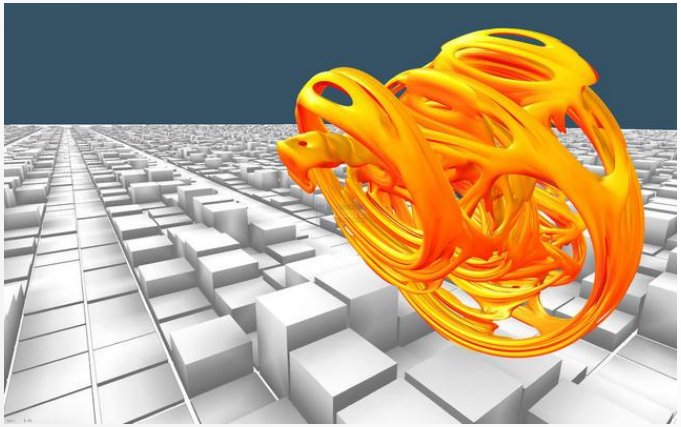


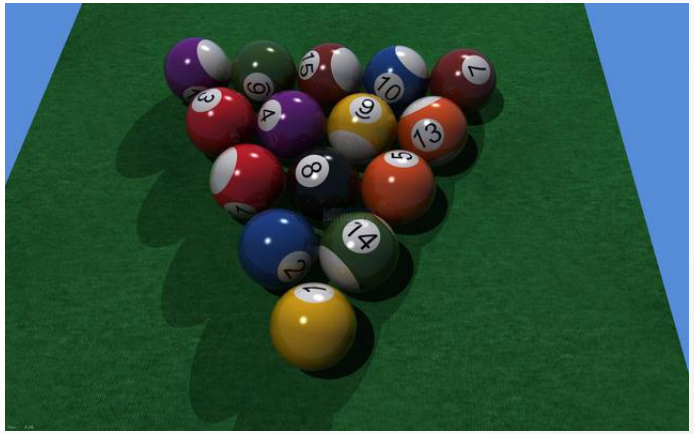
- Projet personnel expérimental ++

Application : Nvidia PhysX

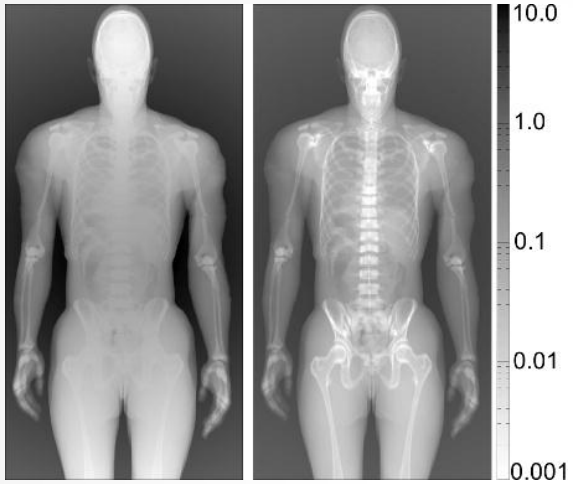
- Habits
- Cheveux
- Particules
- Destruction
- Végétation
- Complex rigid bodies
- Softbodies
- …

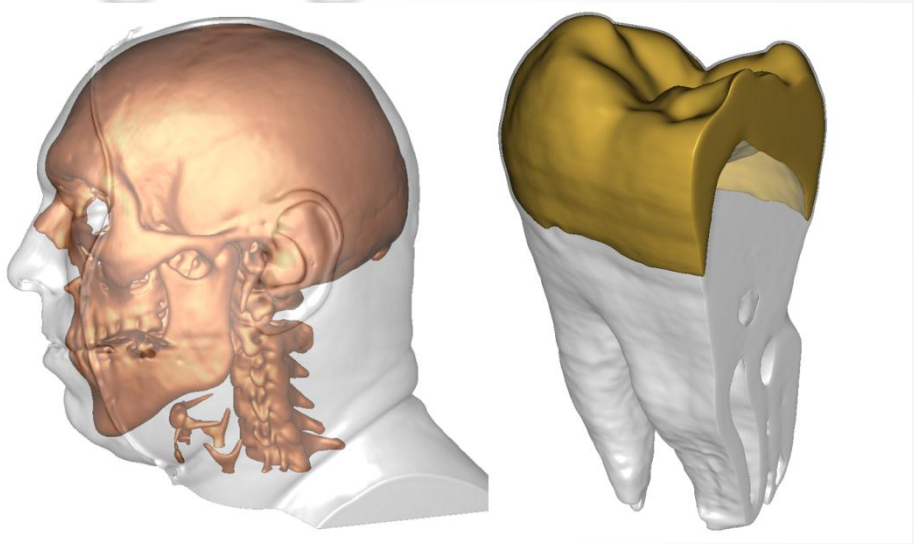
Application : Nvidia OptiX





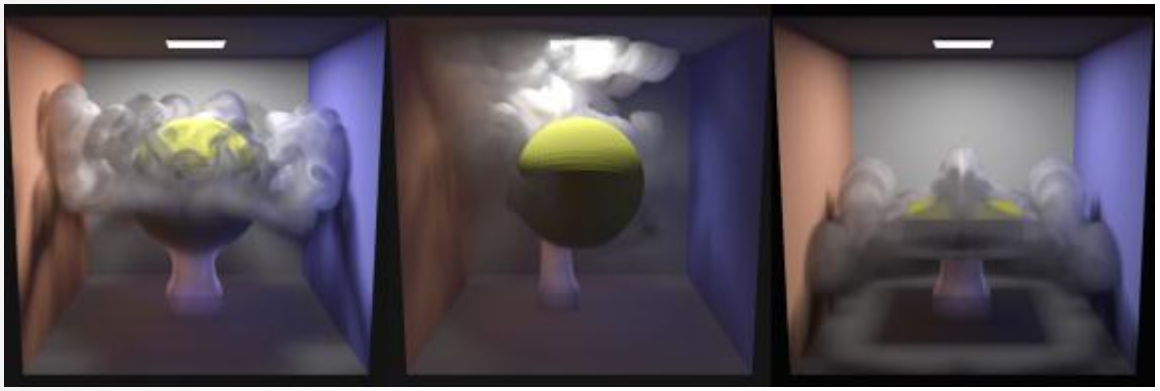
Application : Medical Imaging



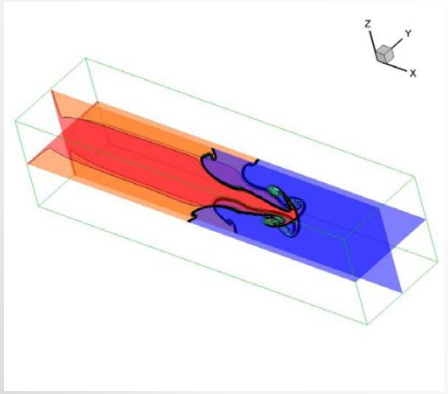


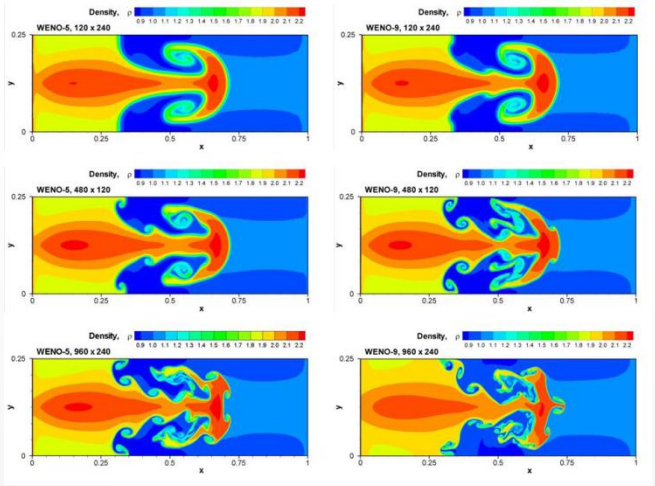
- X-ray transport simulation (x30) :
  - Radiographic projection
  - Computed tomography scans
- High-Quality rendering of Varying Isosurfaces (x68)

Application : Computanional Fluid Dynamics



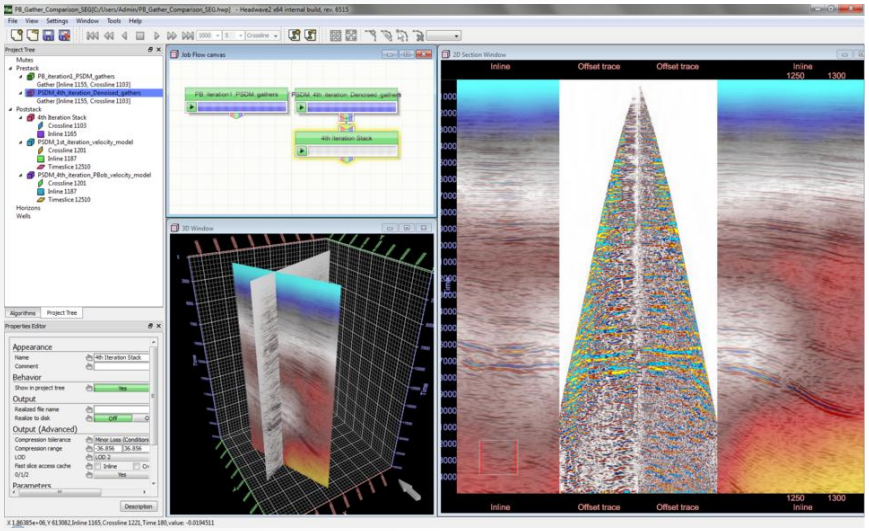
- Simulation de fumée (350 x)



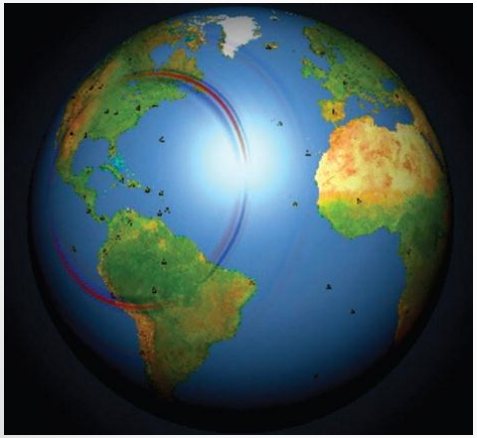


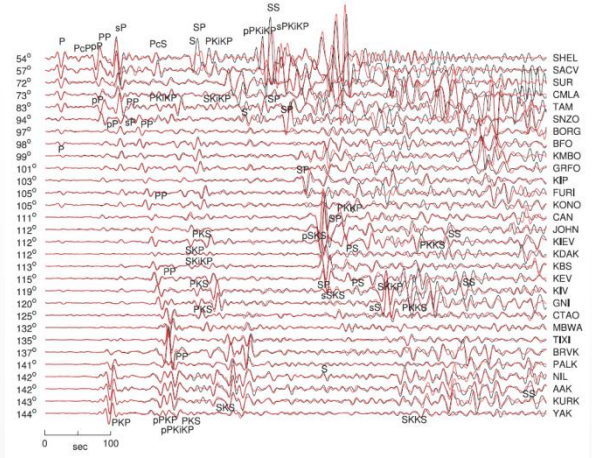
- Simulation de fluide (50 x)

Application : Environmental Science



- Seismic Image Processing (x98)





- Seismic Wave Propagation(x20)

OpenCL

- Modèle de programmation fondamental d'OpenCL très semblables à CUDA
  - Correspondance 1 à 1 entre la plupart des fonctionnalités

- Exemples :

OpenCL	CUDA
• Kernel	• Kernel
• Host program	• Host program
• NDRange (index space)	• Grid
• Work item	• Thread
• Work group	• Block

Conclusions

- Les gpus sont des ordinateurs massivement parallèles :
  - Omniprésent : Les processeurs parallèles les plus réussi de l'histoire
  - Utile : Les développeurs obtiennent des accélérations conséquentes sur des problèmes réels et en peu de temps !
- CUDA est un modèle de programmation parallèle puissant :
  - Hétérogène : Mélange entre programmation séquentielle et parallèle
  - Adaptable-Extensible : Modèle hiérarchique d'exécution de thread
  - Accessible : Plusieurs langages, Oss, vendeurs, librairies, …
- OpenCL aussi
- Enormes possibilités d'innovations…

Questions ?

Références

- « Programming Massively Parallel Processors : A Hands-on Approach » de David B. Kirk & Wen-mei W. Hwu
- « CUDA by Example : An Introduction to General-Purpose GPU Programming » de Jason Sanders & Edward Kandrot
- « NVIDIA CUDA C Programming Guide » (Version 3.2)
- http://www.nvidia.com/object/cuda_home_new.html